Learning for Optimization with Virtual Savant

Renzo Massobrio

Programa de Doctorado en Ingeniería Informática
Escuela de Doctorado
Universidad de Cádiz

Cádiz – España
Mayo de 2021

Learning for Optimization with Virtual Savant

Renzo Massobrio

Tesis de Doctorado presentada al Programa de Doctorado en Ingeniería Informática de la Escuela de Doctorado de la Universidad de Cádiz, como parte de los requisitos necesarios para la obtención del título de Doctor en Ingeniería Informática.

Directores:

Prof. Bernabé Dorronsoro

Prof. Sergio Nesmachnow

Cádiz – España

Mayo de 2021

Acknowledgements

I would like to thank Sergio for his remarkable and tireless support. His guidance and encouragement were fundamental from the start. With his advice, Sergio has sparked my interest in research and profoundly shaped my academic career. Moreover, he has always sought opportunities for me to grow, and for that, I am also grateful.

I would also like to extend my thanks to Bernabé for his outstanding support. It was a pleasure for me to be under his supervision. Bernabé always provided me with invaluable advice and I have learned a great deal from my experience working with him. Besides his role as a supervisor, Bernabé made sure I always felt at home during my time overseas. For that, I am very grateful to both Bernabé and Patricia for welcoming me to their home.

Another thanks goes to Juan Carlos for contributing with parts of the work reported in this thesis but, more importantly, for making my time in Cádiz so much more enjoyable.

Thanks are also in order for my friends in Uruguay and each of the amazing people I have met throughout these years abroad. I will not list them here; each one of them already knows how grateful I am.

A special thanks goes to my father, mother, and sister, for their love, support, and patience while I was away from home during these years.

Finally, I would like to thank Fundación Carolina, Agencia Nacional de Investigación e Innovación (ANII, Uruguay), Universidad de Cádiz, and Universidad de la República for the funding provided for this Ph.D.

ABSTRACT

Optimization problems arising in multiple fields of study demand efficient algorithms that can exploit modern parallel computing platforms. The remarkable development of machine learning offers an opportunity to incorporate learning into optimization algorithms to efficiently solve large and complex problems. This thesis explores Virtual Savant, a paradigm that combines machine learning and parallel computing to solve optimization problems. Virtual Savant is inspired in the Savant Syndrome, a mental condition where patients excel at a specific ability far above the average. In analogy to the Savant Syndrome, Virtual Savant extracts patterns from previously-solved instances to learn how to solve a given optimization problem in a massively-parallel fashion. In this thesis, Virtual Savant is applied to three optimization problems related to software engineering, task scheduling, and public transportation. The efficacy of Virtual Savant is evaluated in different computing platforms and the experimental results are compared against exact and approximate solutions for both synthetic and realistic instances of the studied problems. Results show that Virtual Savant can find accurate solutions, effectively scale in the problem dimension, and take advantage of the availability of multiple computing resources.

Keywords:
machine learning, optimization, Virtual Savant, next release problem, heterogeneous computing scheduling problem, bus synchronization problem.

RESUMEN

Los problemas de optimización que surgen en múltiples campos de estudio demandan algoritmos eficientes que puedan explotar las plataformas modernas de computación paralela. El notable desarrollo del aprendizaje automático ofrece la oportunidad de incorporar el aprendizaje en algoritmos de optimización para resolver problemas complejos y de grandes dimensiones de manera eficiente. Esta tesis explora Savant Virtual, un paradigma que combina aprendizaje automático y computación paralela para resolver problemas de optimización. Savant Virtual está inspirado en el Síndrome de Savant, una condición mental en la que los pacientes se destacan en una habilidad específica muy por encima del promedio. En analogía con el síndrome de Savant, Savant Virtual extrae patrones de instancias previamente resueltas para aprender a resolver un determinado problema de optimización de forma masivamente paralela. En esta tesis, Savant Virtual se aplica a tres problemas de optimización relacionados con la ingeniería de software, la planificación de tareas y el transporte público. La eficacia de Savant Virtual se evalúa en diferentes plataformas informáticas y los resultados se comparan con soluciones exactas y aproximadas para instancias tanto sintéticas como realistas de los problemas estudiados. Los resultados muestran que Savant Virtual puede encontrar soluciones precisas, escalar eficazmente en la dimensión del problema y aprovechar la disponibilidad de múltiples recursos de cómputo.

Palabras claves:
aprendizaje automático, optimización, Savant Virtual, problema del próximo lanzamiento, planificación en sistemas de cómputo heterogéneos, problema de sincronización de autobuses.

Acronyms

0/1-KP 0/1 Knapsack Problem xiv, 2, 3, 4, 35, 41, 45, 47, 48, 55, 62, 64, 119, 121

ANN Artificial Neural Network 6, 38, 122

BLAS Basic Linear Algebra Subprograms 19, 24

BSP Bus Synchronization Problem xiv, xv, 2, 3, 4, 23, 41, 43, 91, 92, 93, 94, 96, 97, 98, 101, 102, 103, 105, 106, 107, 109, 110, 111, 113, 114, 116, 119, 120, 121, 122, 139

DQN Deep Q Network 37, 41

EA Evolutionary Algorithm 6, 13, 14, 69, 70, 96, 97, 100, 105, 106, 109, 110, 111, 113, 114, 116, 120, 122, 139

GA genetic algorithm 32, 40, 41, 70

GP Genetic Programming 32, 41

HCSP Heterogeneous Computing Scheduling Problem xiv, 2, 3, 4, 41, 52, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 86, 87, 89, 116, 119, 120, 121, 122

HPC High Performance Computing 17, 67, 68

LAPACK Linear Algebra PACKage 19

LS local search 13, 14, 39, 53, 54, 61, 62, 70, 72, 73, 74, 75, 76, 77, 80, 81, 82, 84, 86, 100, 106, 109, 110, 113, 114, 116, 122

MA memetic algorithm 70

MCT Minimum Completion Time 70

MILS Multi-start Iterated Local Search 95

MKL Math Kernel Library 19, 24, 26, 27, 28, 29, 30

MPI Message Passing Interface 18, 74

MPNN Message Passing Neural Networks 37, 41

NRP Next Release Problem xiii, xiv, 2, 4, 35, 41, 45, 46, 47, 48, 49, 50, 52, 53, 55, 56, 58, 64, 116, 119, 120, 121, 123

PN Pointer Network 35, 36, 41

RBF Radial Basis Function 21, 24, 53, 57, 71

RE Requirements Engineering 45, 46, 48, 64

RF Random Forest 22, 23, 97, 98, 111, 116, 120, 122

RNN Recurrent Neural Network 35, 41

SAT propositional satisfiability problem 37, 41

SIMD Single instruction, Multiple data 19

SOM Self Organizing Map 39, 41

SPO Smart "Predict, then Optimize" 34, 41

SVM Support Vector Machine xiv, 3, 19, 21, 24, 30, 39, 40, 53, 56, 57, 58, 59, 60, 71, 72, 74, 75, 97, 116, 119, 120, 122

TSP Traveling Salesman Problem 34, 35, 36, 41

VRP Vehicle Routing Problem 33, 36, 123

VS Virtual Savant vii, xiii, xiv, xv, 2, 3, 4, 5, 6, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 23, 31, 35, 39, 40, 41, 43, 45, 48, 49, 50, 52, 53, 55, 56, 58, 59, 60, 61, 62, 63, 64, 65, 67, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 86, 87, 88, 89, 91, 96, 97, 98, 102, 105, 106, 107, 109, 110, 111, 113, 114, 116, 119, 120, 121, 122, 123, 139

Contents

Chapter 1

Introduction

The increasing complexity of optimization problems arising in different fields of study requires algorithms that demand large computing resources (Ausiello et al., 2012). Simultaneously, parallel computing has become a key piece in scientific computing, as it provides the resources needed to solve complex real-world problems that cannot be addressed using classic sequential systems (Golub and Ortega, 2014). Consequently, widespread parallel architectures have led to an increase in the adoption of parallel algorithms that can take advantage of the availability of multiple computing resources.

Software developers need to implement parallel programs to take profit from current architectures. This requires highly-skilled programmers that can design parallel programs from scratch or redesign legacy sequential implementations to profit from modern parallel architectures. Thus, there is an increased interest in techniques that can automatically generate elastic programs that can fully exploit highly-parallel computer platforms and scale in the number of computing resources (Darte et al., 2012). The current growing interest in machine learning techniques comes at hand to deal with this problem.

The fields of optimization and machine learning are closely related. However, the vast majority of research has explored one direction of this relationship, i.e., optimization applied to machine learning techniques (e.g., parameter optimization in machine learning models, feature selection problems) (Sra et al., 2012). The inverse, i.e., applying machine learning to solve optimization problems, while explored (Vlastelica et al., 2020; Vinyals et al., 2015), still has plenty of room for contribution.

This thesis deals with Virtual Savant (VS), a novel paradigm that takes advantage of machine learning and parallel computing to address complex optimization problems (Pinel et al., 2013). VS is inspired in the Savant Syndrome, a mental condition where patients excel at certain abilities far above the average. In analogy to the Savant Syndrome, VS uses machine learning to find patterns that allow solving the problem at hand. These patterns are learned from a set of previously-solved instances of the problem. Due to its design, VS can be executed in massively-parallel computing architectures, significantly reducing execution times and effectively scaling in the problem instance.

The main goal of the research reported in this thesis is the implementation of VS, its application to multiple optimization problems, and its evaluation on different computing platforms. The main contributions of this work are:

1. A comprehensive review of the related literature in the automatic generation of parallel programs and the synergy between machine learning and optimization.
2. A thorough definition of the VS workflow and its implementation.
3. The application and evaluation of VS to the Next Release Problem (NRP), a combinatorial optimization problem arising in software engineering that is modeled as a 0/1 Knapsack Problem (0/1-KP).
4. The application of VS to solve the Heterogeneous Computing Scheduling Problem (HCSP) and the study of its scalability in different computing platforms and with varying problem sizes.
5. The application of VS to the Bus Synchronization Problem (BSP), a complex combinatorial optimization problem arising in public transportation networks, and the evaluation over synthetic and real-world problem instances.

The work related to this thesis has led to the publication of several journal and conference articles. A list of these publications and on-going submissions, along with a brief description of their contents, is presented next.

- "Generación automática de programas: Savant Virtual para el problema de la mochila" presented at XI Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados, Salamanca, Spain (Massobrio et al., 2016). This article outlined the first application of VS to the 0/1-KP.

- "Automatic program generation: Virtual Savant for the knapsack problem" presented at International Workshop on Optimization and Learning: Challenges and Applications, Alicante, Spain (Massobrio et al., 2018b). This article extended the study of the behavior of VS when solving the 0/1-KP.
- "Support Vector Machine Acceleration for Intel Xeon Phi Manycore Processors" presented at Latin America High Performance Computing Conference, Buenos Aires, Argentina (Massobrio et al., 2018c). This article presented xphi-LIBSVM, a parallel implementation of the popular LIBSVM library for Support Vector Machines (SVMs), specifically adapted to the Intel®Xeon Phi™ architecture.
- "Virtual Savant for the Heterogeneous Computing Scheduling Problem" presented at International Conference on High Performance Computing & Simulation, Orléans, France (Massobrio et al., 2018a). This article outlined the application of VS to the HCSP.
- "Virtual Savant for the Knapsack Problem: learning for automatic resource allocation" published in the Proceedings of the Institute for System Programming of the Russian Academy of Sciences (Massobrio et al., 2019). This article studied the scalability of VS when solving the 0/1-KP.
- "Parallel Virtual Savant for the Heterogeneous Computing Scheduling Problem" published in the Journal of Computational Science (de la Torre et al., 2020). This article presented a parallel implementation of VS for the HCSP.
- "Urban Mobility Data Analysis for Public Transportation Systems: A Case Study in Montevideo, Uruguay" published in Applied Sciences (Massobrio and Nesmachnow, 2020). This article presented a study on the public transportation network of Montevideo, Uruguay, that was used to generate realistic problem instances to evaluate VS on the BSP.
- "Evolutionary approach for bus synchronization" presented at Latin America High Performance Computing Conference, Turrialba, Costa Rica (Nesmachnow et al., 2020). This article presented an evolutionary algorithm for the BSP, which was later used to solve the same problem with VS.

- "Virtual Savant: learning for optimization" presented at the Learning Meets Combinatorial Algorithms workshop of the 34^{th} Conference on Neural Information Processing Systems, Vancouver, Canada (Massobrio et al., 2020). This article presented preliminary results of the application of VS to the BSP.
- "Virtual Savant as a generic learning approach applied to the basic independent Next Release Problem" under review for possible publication in Applied Soft Computing. This article presents the application of VS to solve the NRP.
- "Learning to optimize timetables for efficient transfers in public transportation systems" submitted to the Special Issue on Intelligent solutions for efficient logistics and sustainable transportation of Applied Soft Computing. This article presents the application of VS to solve the BSP.

The remainder of this thesis is structured as follows. Chapter 2 outlines the VS paradigm, including its motivation, the idea that inspires the method, the complete workflow, and implementation details, including the implementation and evaluation of xphi-LIBSVM, a parallel implementation of the LIBSVM library that is used in VS. Then, Chapter 3 provides a literature review on the automatic generation of programs that can run in parallel architectures and on the synergy between machine learning and optimization. The first application of VS is presented in Chapter 4, which outlines how VS can be used to solve the NRP modeled as a 0/1-KP. A thorough experimental evaluation is presented using a large set of problem instances with varying size and difficulty. Afterward, the application of VS to the HCSP is presented in Chapter 5. The experimental evaluation for this problem focused on studying the scalability of VS both in terms of the problem size and in the use of computational resources. The application of VS to the BSP is presented in Chapter 6, where VS is evaluated over synthetic and realistic problem instances. Finally, the conclusions and main lines of future work are outlined in Chapter 7.

Chapter 2

Learning for Optimization: the Virtual Savant Paradigm

This chapter presents the VS paradigm. First, the motivation that drives VS is outlined in Section 2.1. Then, Section 2.2 describes the Savant Syndrome, which inspires the VS model. After that, Section 2.3 outlines the conceptual framework of VS. Then, Section 2.4 provides details regarding the implementation of the VS paradigm. Finally, Section 2.5 presents the experimental evaluation of xphi-LIBSVM, a parallel version of a machine learning library that was implemented to execute VS in many-core computing platforms.

2.1 Motivation

Optimization consists in finding a solution over a defined set that optimizes a given objective function. In particular, the subfield of combinatorial optimization deals with finding solutions over finite sets. Combinatorial optimization problems arise in a plethora of domains, e.g., logistics, supply-chain management, software engineering, transportation. Consequently, research on algorithms that can solve optimization problems is vast in the literature and includes exact approaches (Woeginger, 2003) and approximation algorithms that find near-optimal solutions (Blum and Roli, 2003; Aarts and Lenstra, 2003).

In the case of complex optimization problems or very large problem instances, which are ubiquitous these days, the execution time and computational performance of algorithms are of the essence, as they directly affect

whether the computed solutions can be put into practice. Due to the increasing availability of parallel computing platforms, parallel algorithms have become a key piece of scientific computing in general and of optimization research in particular.

At the same time, the field of machine learning has had an astounding development, with new methods and applications emerging at a remarkably fast pace (Jordan and Mitchell, 2015). Machine learning algorithms build a model based on sample data and make predictions without being explicitly programmed to do so. Thus, they can take advantage of historical data to make accurate predictions in different domains, e.g., computer vision, speech recognition, medical diagnosis, bioinformatics.

The fields of optimization and machine learning are intrinsically intertwined and there is an increasing synergy between both research communities (Bennett and Parrado, 2006). On the one hand, optimization plays a key role within machine learning, since most learning problems reduce to optimization problems. In this sense, several notions of optimization theory and algorithms have been applied to machine learning methods, e.g., for effective parameter tuning and feature selection (Sra et al., 2012; Bottou et al., 2018). On the other hand, the application of machine learning methods to solve optimization problems, while existing, is not nearly as prevalent in the literature (Bengio et al., 2021).

VS, the paradigm explored in this thesis, aims to take advantage of machine learning techniques to automatically generate programs that solve complex optimization problems and can be run in parallel, making use of the multiple resources available in modern computing platforms. The idea that inspires this paradigm is presented next.

2.2 Inspiration: the Savant Syndrome

Natural computing refers to the process of extracting ideas from nature to create computational tools that solve complex problems (de Castro, 2006). Biologically inspired—or *bioinspired*—techniques have been applied to a wide range of complex optimization and decision-making problems (Olariu and Zomaya, 2005). The power of these techniques lies in their capability to explore complex search spaces with little to no problem-specific knowledge. Examples of bioinspired techniques include: Artificial Neural Networks (ANNs), inspired

by the nervous system; Evolutionary Algorithms (EAs), inspired by evolutionary biology; Swarm Intelligence, inspired by the collective behavior of groups of organisms; and Artificial Immune Systems, inspired by theoretical and experimental immunology. The paradigm explored in this thesis, Virtual Savant, may also be considered as bioinspired since it emulates a natural phenomenon: the *Savant Syndrome*.

The Savant Syndrome is a rare mental condition where patients with significant mental disabilities develop certain abilities far above what would be considered average (Treffert, 2006). Patients with Savant Syndrome—known as *savants*—usually excel at a single specific activity, generally related to memory, rapid calculation, or artistic abilities. The underlying thought processes of savants are not yet fully understood by researchers. However, the main hypotheses state that savants learn through pattern recognition (Pring, 2005; Heaton and Wallace, 2004). This mechanism allows them to solve problems without understanding their underlying principles. For instance, some patients can enumerate large prime numbers or discriminate between prime and non-prime numbers, without fully understanding what a prime number is.

Savant Syndrome often manifests along with other mental disabilities. Reports suggest that half of the people with Savant Syndrome have autistic disorder while the other half endure other forms of developmental disability or brain injury (Treffert, 2009). While rare, estimates suggest that up to one in ten patients with autism present some degree of Savant Syndrome, with male patients outnumbering females by an approximate 6:1 ratio (Rimland, 1978).

Reported skills of savants are limited to a rather narrow list comprised of: music abilities, including extraordinaire performance and the ability to play multiple instruments; art abilities, including prodigious drawing, painting, and sculpting; calendar calculating, including complex date arithmetic (e.g., naming the day of the week corresponding to a given date, counting the number of seconds between two very distant dates); mathematics, including lightning calculations and prime number enumeration; and spatial skills, including mapmaking, accurate distance estimation, and complex route planning. Generally, savants show a single special skill, but some patients excel at multiple skills simultaneously (Rimland and Fein, 1988). Regardless of the skill, prodigious memory is an ability exhibited across all patients with Savant Syndrome. Treffert (2009) suggested that savants' skills exhibit a pattern of "replication to improvisation to creation". For instance, musical savants may begin accurately

playing back complex classical music, then move forward to improvise over the original piece, and finally, even create entire new pieces on their own.

Savants cannot usually give insights into how they perform their unique skills. Since the condition often presents along with other mental disabilities, understanding the thought processes of savants is remarkably challenging. Snyder (2009) argued that the skills of savants are the result of privileged access to raw, less-processed sensory information. This information is available to every person but is usually inaccessible due to top-down inhibition of the brain. The author showed that some of these skills can be artificially induced in otherwise average people by temporally inhibiting parts of the brain using magnetic pulses. The author concluded that savants have a "tendency to concentrate more on the parts than on the whole... [which] offers advantages for particular classes of problem-solving...".

Some investigations have compared savants and non-savants with talents in the same domain. A definitive answer to whether savants use the same or different cognitive strategies than regular people is yet to be found. Nonetheless, the highly-efficient computational abilities of savants are claimed to be different from existing methods (Pring, 2005). For instance, savants with calendrical calculation skills lack the reading comprehension to understand existing algorithms to compute dates. Thus, savants are believed to combine rote memorization with implicit learning to solve problems by allowing structured regularities in the input to emerge. Supporting this hypothesis, Heaton and Wallace (2004) argued that the thought processes of savants are influenced by pattern recognition and the construction of artificial grammars, providing building blocks for knowledge acquisition. Thus, savants tend to process individual features—which are easier to remember—to build highly structured information. Consequently, concentration, repetition, and practice were found to be important to maintain and reinforce the skills of savants.

Similarly, Hermelin et al. (1999) argued that the responses of savants are based on the extraction of intrinsic rules and regularities from the material given as input. Through a process of transformation, savants can derive knowledge of a global system from a collection of single instances. However, the authors highlighted that the answers of savants to a specific task are rarely 100% accurate. In their study of an artistic savant, the authors noted that paintings of memorized scenes, although precise, were not exact replicas. In the studied paintings, several features were added and others omitted, and changes in

size and color transformations were also present. Similar results were assessed when comparing the accuracy of artistic savants to that of average children when drawing a memorized model (Hermelin and O'Connor, 1990). The conjecture that savants answers are not exact but rather probabilistic can also be found in studies involving calendrical savants. Mottron et al. (2006) performed several experiments with a calendrical savant, one of which consisted of naming the day of the week corresponding to a given date. A subset of dates was asked repeatedly in two independent sessions. The answers provided by the studied savant were not consistent, exhibiting a different error pattern and response time distribution each session.

Another factor that may contribute to the exhibited skills of savants is related to parallel processing. Treffert (2013) described the case of Leslie, a blind musical savant with extraordinaire abilities to play back and even improvise over musical pieces. In one experiment, Leslie was asked to play along with another musician a previously-unheard piece of music. After three seconds of hearing the musician play, Leslie started playing along without interruptions. This behavior suggests that Leslie was parallel processing: simultaneously hearing the tune that was being played, processing what he heard, and finally playing along with the musician. The ability to process in parallel calls into question the IQ scores assigned to Leslie (in the 35-55 range). Thus, the author suggested that this complex performance evidences "that more than a single 'intelligence' was at work". Mottron et al. (2009) suggested that the independent cognitive processes of savants may allow integrating patterns in parallel without information loss. Similarly, Yamaguchi (2009) speculated about some parallel algorithms that may explain the abilities of a pair of savant twins with extraordinary prime number identification skills.

Summarizing, although Savant Syndrome is a complex phenomenon yet to be fully understood, literature agrees on a few key factors: i) savants are unaware of the algorithms and fundamental principles related to the problem they address; ii) savants use some form of pattern recognition on the input data; iii) savants aggregate individual pieces to derive global knowledge; iv) repetition and learning enhance the abilities of savants; v) the outcomes of savants are probabilistic, giving different responses to the same input; and vi) savants make use of some form of parallelization in their thought processes. These aspects of the frame of thought of savants are the key concepts driving the design of VS, which is presented next.

2.3 Virtual Savant

This section presents VS. First, an overview of the technique is outlined and then, each of its phases is described. Finally, the parallel design of VS is presented.

2.3.1 Overview

VS is a novel technique, inspired by the Savant Syndrome, that aims to learn how to solve a given optimization problem (Pinel et al., 2013). As an analogy to the Savant Syndrome, VS proposes using machine learning techniques to find patterns that allow solving the problem at hand. These patterns are learned using machine learning from a set of previously-solved instances of the problem. Solutions used for training are computed by one (or several) reference algorithm(s) for the problem. VS does not require knowing the code of the reference algorithms it learns from, in the same way that real-life savants are unaware of the underlying principles related to their skill. The training of VS involves partitioning the problem instance, and like savants, VS can derive global solutions by combining smaller pieces.

Savants can enhance their abilities via repetition and learning. Analogously, VS is able to compute more accurate results by improving the set of solved problem instances used during training. Once the training phase is completed, VS can solve unseen and larger problem instances, without the need of any further retraining. In resemblance to savants, VS is stochastic and does not guarantee computing exact solutions. Thus, multiple VS executions may return different approximate solutions to the problem at hand.

Similarly to real-life savants, which are thought to have parallel processing capabilities, VS can also take advantage of multiple computing resources. Due to its design, each phase in VS can be executed in a massively-parallel fashion. Thanks to parallelism, VS can reduce execution times significantly, find better solutions due to an improved exploration of the search space, and effectively scale in the problem dimension.

2.3.2 Training of VS

VS is trained using previously-solved problem instances. Learning is solely based on the input (i.e., the problem instance) and the output (i.e., the so-

lution) computed by one or several reference algorithms, without the need to know how those algorithms work or how the solutions were computed. In fact, VS can learn from preexisting benchmarks of solved instances, which are frequently available for widely-known optimization problems. Therefore, the first step when applying VS to any optimization problem is to build a dataset of solved problem instances from which a training set is generated for VS to learn from.

One of the key design principles in VS is that it aims to learn relationships between the elements of the problem. The simplest case models variables (univariate analysis), but more complex models (bivariate, multivariate) can be considered as well. For the sake of simplicity, the univariate case is used to outline the VS workflow. The univariate case considers as many training examples as variables in the problem being solved. Thus, each solved problem instance yields as many training examples as variables in the problem being solved. Figure 2.1 outlines the process of transforming a dataset of solved instances into a training set, for a problem with n variables. The problem instance is transformed into multiple training vectors that are included in the training set. Each training vector j is comprised of k features $x_1^j \dots x_k^j$, which are taken from the problem instance and a label y^j that corresponds to the value of variable j in the solution. The main challenge during training, as in most machine learning problems, lies in selecting the instance features that maximize the accuracy of the trained model. This decision is highly problem-specific and involves a process of instantiating the general VS schema to the problem at hand. A discussion on which features to include during training is presented for each case study outlined in Chapters 4–6

Once the training set is generated, a supervised machine learning model is trained over that set. Supervised learning refers to the task of approximating a mapping function that associates an input to an output based on input-output pairs given as examples (Russell and Norvig, 2010). In the case where the output is within a finite set of values, the supervised learning problem is called a classification task.

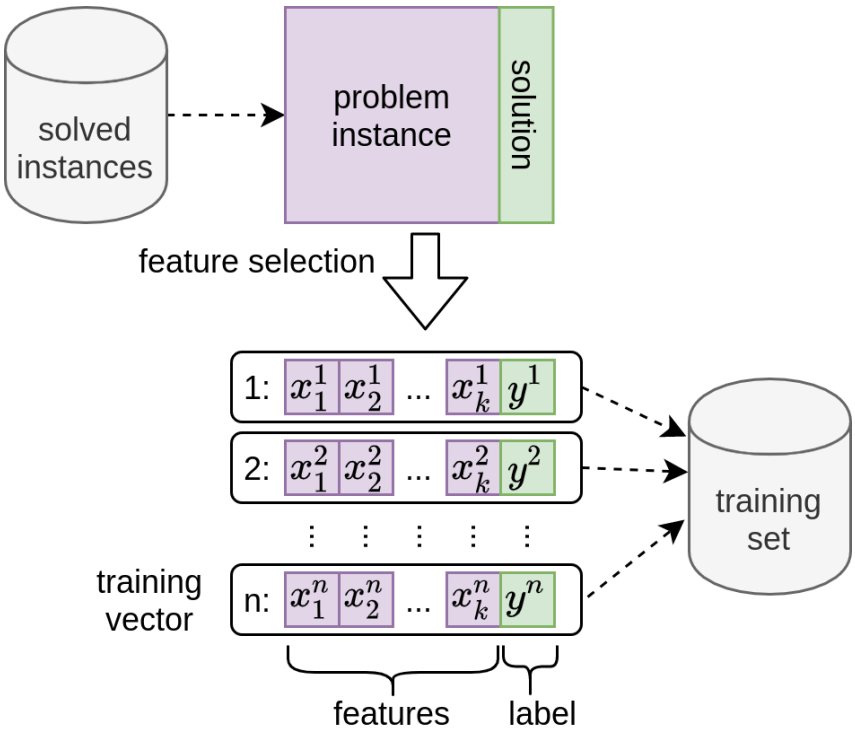


Figure 2.1: VS training set generation.

2.3.3 Execution of VS

Once training is completed, VS can solve new—unseen—problem instances by following a two-phase process comprised of *prediction* and *improvement*. These phases are described next.

2.3.3.1 Prediction phase

In the prediction phase, the trained classifier is used to predict a solution to a new, unseen, problem instance. In the univariate case, each variable in the new problem instance can be predicted independently, thanks to the design followed in the training process of VS. Figure 2.2 outlines the prediction phase of VS. Given an unsolved problem instance, features corresponding to each variable in the problem are extracted in the same way as during the training process. As a result, VS gets as many unlabeled feature vectors as variables in the problem instance. Each of these feature vectors is separately fed to the trained model, which outputs a prediction $\hat{y}^i$ for the variable corresponding to that feature vector. The predicted solution to the problem instance is returned by aggregating the individual predictions of each problem variable.

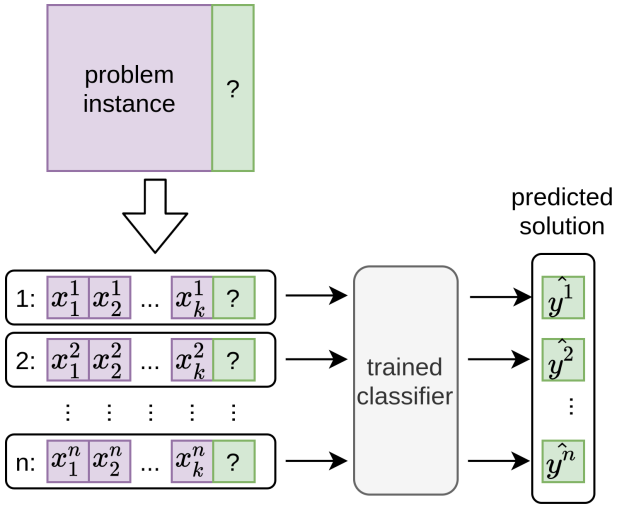


Figure 2.2: VS prediction phase.

Many machine learning libraries implement methods that allow estimating label probabilities for multi-class classification problems (Wu et al., 2004). The output of the trained model in these methods is not a single label but instead a vector of size equal to the number of possible classes, indicating the probability of labeling the given input to each of the possible classes. Figure 2.3 outlines the prediction phase of VS when using this approach. In this case, the output of the prediction phase is a probability distribution $P(\hat{y}^i)$ for each of the i variables in the problem. The predicted solution shown in Figure 2.2 can be built by simply taking the arg max of each vector of probabilities.

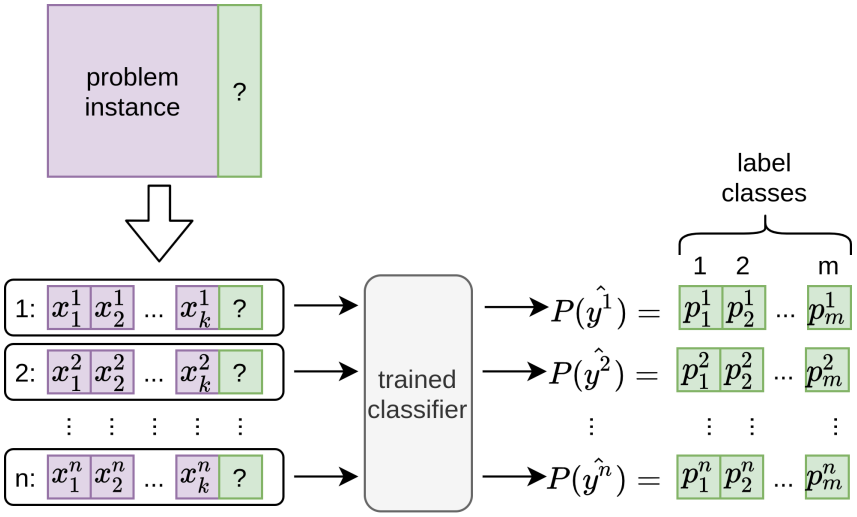


Figure 2.3: VS prediction phase with label probabilities.

2.3.3.2 Improvement phase

During the improvement phase, the predicted solutions are refined using search procedures and heuristics. For instance, a local search (LS) algorithm can be applied to the predicted solutions to improve their quality in terms of the objective function of the optimization problem being solved. Additionally, predicted solutions can be given as input seeds to more complex optimization methods such as EAs and other metaheuristics. When solving problems with constraints, generated solutions may be infeasible due to inaccuracies in the prediction phase or modifications during the improvement phase that may lead to unsatisfied constraints. Thus, corrective functions must be included in the improvement operator to ensure that the returned solution satisfies all problem constraints. In general, the rationale behind VS is to use general optimization strategies during the improvement phase, without relying on problem-specific techniques. By doing so, VS can be applied to solve multiple problems from many different domains.

When using the strategy for the prediction phase that outputs probability distributions of labels for each variable instead of a single prediction, the improvement phase involves generating multiple candidate solutions following those probability distributions $P(\hat{y^i})$. Each of these candidate solutions can be improved using an improvement operator (e.g., LS algorithm, greedy heuristic) or, instead, the whole set can be used to start a population-based strategy (e.g., an EA). The workflow is outlined in Figure 2.4: r candidate solutions are built drawing values using the probability distributions computed during the prediction phase, which are then improved using the improvement operator. The overall best solution found is returned.

2.3.4 Parallelism in VS

Thanks to its design, VS can be run in a massively-parallel fashion. The parallel model of VS can be interpreted as a case of a MapReduce approach (Dean and Ghemawat, 2004). Both phases in VS are subject to parallelism.

In the prediction phase, predictions can be made in parallel by using multiple copies of the same trained classifier, since each element in the problem is learned independently. These classifiers can make their predictions independently of one another, without the need for costly communications. An interesting aspect of the design of VS is that, when there are as many com-

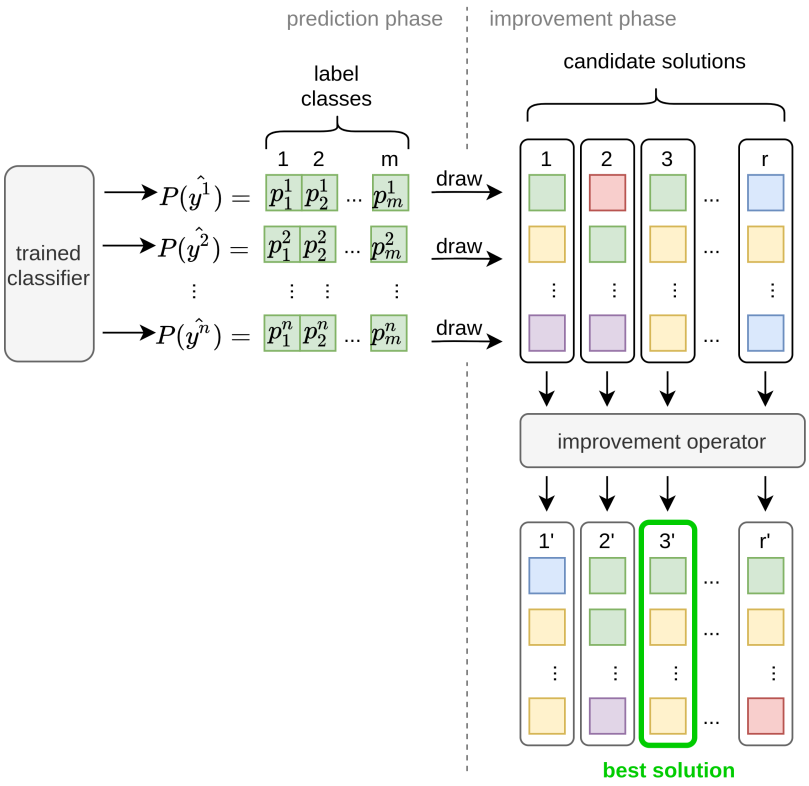


Figure 2.4: VS improvement phase.

puting nodes available as problem variables, it allows keeping nearly-constant response time when increasing the size of the problem. Solving a larger problem instance is as simple as launching more copies of the trained classifier, one for each variable in the problem.

Similarly, the improvement phase is also subject to a high degree of parallelism. After label probabilities are computed for each variable, multiple candidate solutions can be built and improved in parallel. Thus, VS can take advantage of available computing resources to improve its search of the solution space, leading to better solutions. Once again, no communication is required among the parallel processes, since each one is improving a different candidate solution. Nevertheless, sophisticated cooperative models involving communications can also be applied to further improve the computed solutions. The best overall solution computed by the pool of parallel processes is returned as the solution to the problem by VS.

Figure 2.5 outlines the complete workflow of VS when running in parallel. For each of the n variables in the problem instance, a copy of the trained classifier is spawn. Each classifier receives the vector of features corresponding

to one of the problem variables and outputs a vector $P(\hat{y^i})$, indicating the probability of predicting each of the possible m labels for that variable. The first synchronization barrier (shown in red) allows waiting for all classifiers to output their predictions. Once all classifiers have completed their task, candidate solutions are generated by drawing values for each variable according to the probabilities predicted by the classifiers. In the example, r candidate solutions are built, each of which is improved by applying the improvement operator in parallel. The second synchronization barrier is in charge of waiting for all the improvement operators to finish. Finally, all solutions are gathered and the best overall solution is returned.

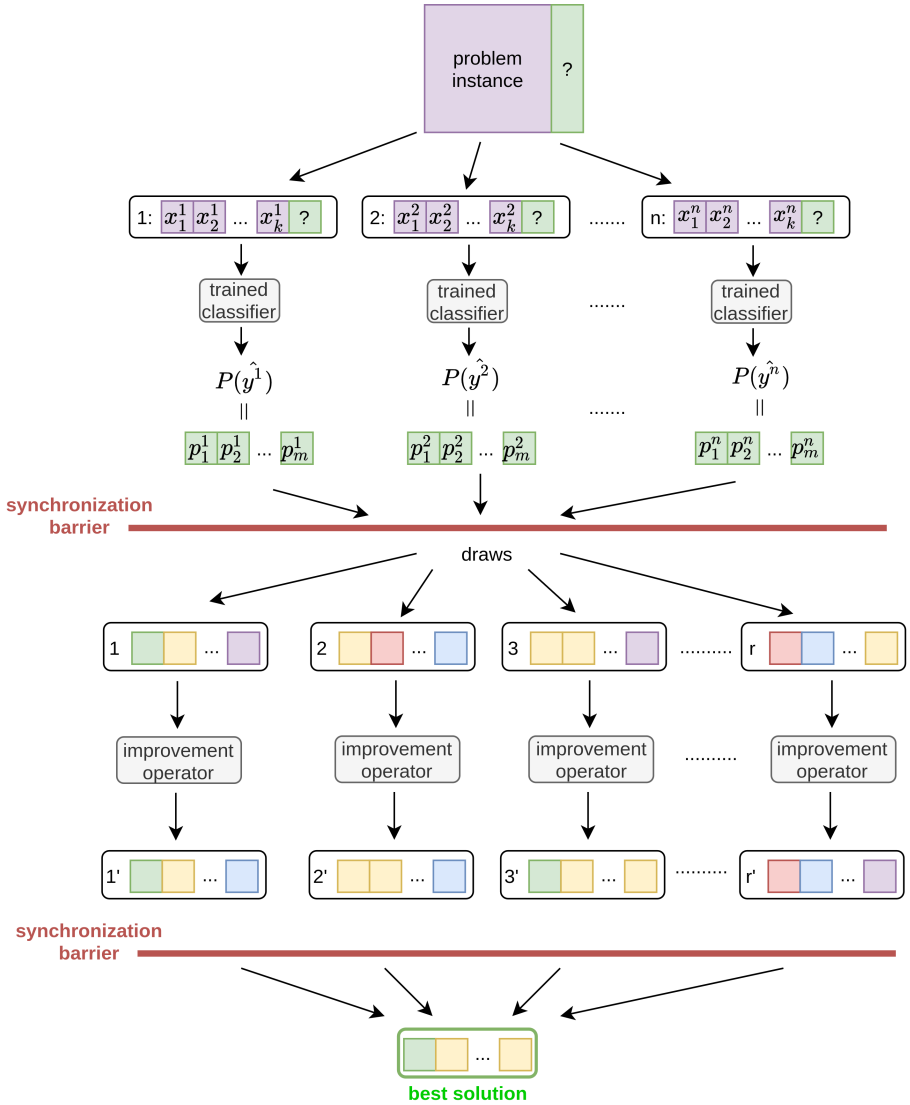


Figure 2.5: Parallel VS workflow.

2.4 Implementation of VS

This section describes the implementation details of VS.

2.4.1 Parallel implementation

Details on the computational platforms and parallel development libraries used for the implementation of VS are outlined next.

2.4.1.1 Computational platforms

Different computational platforms were used to evaluate the performance and scalability of VS when solving the optimization problems considered in this thesis. These platforms included regular desktop PCs, standalone servers, High Performance Computing (HPC) clusters, and the Intel®Xeon Phi™ architecture, which is briefly presented next.

Xeon Phi™ is a brand name given to a series of many-core processors by Intel®. Many-core processors are multi-core processors specially designed for a high degree of parallelism, consisting of tens or thousands of simpler independent cores. The use of many-core processors has been increasing in the past years, with extensive applications in embedded systems and HPC platforms. The Xeon Phi™ family of processors was initially designed as an add-on PCIe card that could be connected to a standard CPU and used for computing-intensive tasks. A second generation of Xeon Phi™ products, with codename Knights Landing, was announced in June 2013. The main difference with its prior generation is that Knights Landing are stand-alone processors that can boot an off-the-shelf operating system. Therefore, Knights Landing avoids the bottlenecks in PCIe communications—which are inherent in coprocessors—and provides a powerful HPC platform in a standard CPU form factor.

Sodani et al. (2016) presented an overview of the Knights Landing architecture, which consists of 38 physical tiles: at most 36 are active and the remaining two are used for recovery purposes. Each tile has two cores, two vector processing units per core, and a shared 1MB L2 cache. The processing cores derive from the Intel®Atom™ core microarchitecture but incorporate several modifications specially designed to suit HPC workloads. Each core supports up to four hardware contexts or threads through Hyper-Threading

techniques. All these features make the Knights Landing architecture a good candidate for HPC tasks, without requiring any special way of programming other than the standard CPU programming model, and even having decent support for serial legacy code.

2.4.1.2 Parallel tools and implementation libraries

The tools and libraries used for the parallel implementation of VS are outlined next according to their level of abstraction.

2.4.1.2.1 Message Passing Interface (MPI) is a message-passing standard that supports a wide variety of parallel computing platforms (Message Passing Interface Forum, 2015). MPI defines a core library of routines to help developers of parallel applications and has become the de facto standard for parallel computing in distributed-memory systems. The MPI interface provides virtual topology, synchronization, and communication functionalities between a set of processes that are mapped to computing resources (e.g., nodes, servers), among several other functionalities. MPI was used in this thesis for the parallel implementation of VS in distributed-memory architectures.

2.4.1.2.2 OpenMP is a specification of compiler directives and library routines to implement high-level parallelism in Fortran and C/C++ programs (Dagum and Menon, 1998). OpenMP can significantly improve the scalability of shared-memory parallel applications. Developers need to define specific directives in their sequential code to indicate how the program should be parceled out among the individual processors in a symmetric multiprocessing computing platform. Then, the compiler that supports OpenMP transforms the sequential code into an executable that takes advantage of the multiple processors available. OpenMP standardizes this notation to be able to support different hardware platforms. OpenMP was used in this thesis for the parallel implementation of VS in shared-memory architectures.

2.4.1.2.3 Intel®C++ Compiler is part of the Intel®Parallel Studio XE suite. The compiler incorporates many optimizations to take advantage of specific processor features, such as the number of available cores and wider vector registers, to speed up computations. Intel®C++ compiler has broad

support for current and previous C and C++ standards, including full support for C++11 and C99. Furthermore, it supports integration with OpenMP for parallel implementations. Intel®C++ compiler was used in this thesis to implement xphi-LIBSVM: a parallel version of the LIBSVM machine learning library, specifically adapted to the Intel®Xeon Phi™ architecture.

2.4.1.2.4 Intel®Math Kernel Library (MKL) supports a series of optimized and threaded mathematical functions to take advantage of the architecture of Intel®processors to solve large problems. MKL performs a hardware check on runtime and selects suitable functions to improve execution time through instruction-level and register-level Single instruction, Multiple data (SIMD) parallelism (Wang et al., 2014). Intel®MKL also incorporates thread-safe functions to speedup computations using OpenMP. The library provides Basic Linear Algebra Subprograms (BLAS) and Linear Algebra PACKage (LAPACK) routines, fast Fourier transforms, vectorized math functions, random number generation functions, and many other features. Specific functions of the Intel®MKL were used in this thesis for the implementation of the xphi-LIBSVM library.

2.4.2 Machine learning classifiers

The VS model is agnostic in terms of which classifier to use and the decision is left to the practitioner. Descriptions of the two classifiers used in the implementations of VS developed in this thesis are presented next.

2.4.2.1 Support Vector Machines

SVMs are supervised machine learning models used for classification and regression analysis. Used in many different fields of study, SVMs are considered as a standard out-of-the-box classifier due to their good performance (James et al., 2014). SVMs are an extension of support vector classifiers, which are themselves a generalization of an even simpler classifier known as the maximal margin classifier, which is described next.

Consider n training observations in a p-dimensional space, $x^1=(x_1^1, x_2^1, \ldots, x_p^1), \ldots, x^n=(x_1^n, x_2^n, \ldots, x_p^n)$, corresponding to a binary classification problem, i.e., each observation has a corresponding label $y^1 \ldots y^n \in \{-1, 1\}$ where -1 and 1 are the two possible classes. Given a test observation, i.e., a feature

vector $x^*=(x_1^*, x_2^*, \dots x_p^*)$ for which the label y^* is unknown, the goal is to build a classifier that can correctly classify the vector into one of the two possible classes. The maximal margin classifier does so by finding a hyperplane that separates the training observations in the p-dimensional space according to their class labels, so that each observation of one of the classes lies in one side of the hyperplane and the ones corresponding to the other class lie in the other side. If training data is linearly separable, then an infinite number of possible hyperplanes exist that can separate the observations according to their classes. The maximal margin classifier consists in finding the separating hyperplane that has the largest minimum distance to the training observations. Once the hyperplane is set, a new—unseen—observation can be classified depending on which side of the hyperplane it lies on. The mathematical formulation for the maximal margin classifier is presented in Equations 2.1.

$$\max_{\beta_0,\beta_1,\dots,\beta_p,M} \quad M \tag{2.1a}$$

$$\text{subject to} \sum_{j=1}^{p} \beta_j^2 = 1, \tag{2.1b}$$

$$y^i(\beta_0 + \beta_1 x_1^i + \beta_2 x_2^i + \dots + \beta_p x_p^i) \geq M \quad \forall i = 1, \dots, n. \tag{2.1c}$$

The goal of the optimization problem is to find the hyperplane defined by $\beta_0, \beta_1, \dots, \beta_p$ that maximizes M (Equation 2.1a), which is the positive margin that ensures observations fall in the correct side of the hyperplane at a distance of at least M. From Equations 2.1b and 2.1c it can be derived that the perpendicular distance for the i^{th} observation is given by $y^i(\beta_0 + \beta_1 x_1^i + \beta_2 x_2^i + \dots + \beta_p x_p^i)$. Constraint 2.1c ensures that all training vectors fall on the correct side of the hyperplane according to their label.

In many cases, classification problems are not linearly separable, i.e., there is no solution to the previous optimization problem with $M > 0$. In this case, a hyperplane that almost separates the classes may be computed. This idea is the basis for the support vector classifier, which finds a separating hyperplane with a soft margin, allowing a certain number of training vectors to fall on the wrong side of the hyperplane. The optimization problem corresponding to the support vector classifier is presented in Equations 2.2.

$$\max_{\beta_0,\beta_1,\ldots,\beta_p,M} \quad M \tag{2.2a}$$

$$\text{subject to } \sum_{j=1}^{p} \beta_j^2 = 1, \tag{2.2b}$$

$$y^i(\beta_0 + \beta_1 x_1^i + \beta_2 x_2^i + \ldots + \beta_p x_p^i) \geq M(1 - \epsilon_i) \tag{2.2c}$$

$$\epsilon_i \geq 0, \sum_{i=1}^{n} \epsilon_i \leq C. \tag{2.2d}$$

The optimization problem is similar to the one corresponding to the maximal margin classifier, but constraint 2.2c incorporates the notion of soft margins using variables $\epsilon_1, \ldots, \epsilon_n$, bounded by parameter C, which limits the number and magnitude of violations to the margin (Equation 2.2d). If $C = 0$ the support vector classifier reduces to a maximal margin classifier. The support vector classifier can be formulated by Equation 2.3, where $\langle a, b \rangle = \sum_{i=1}^{p} a_i b_i$ is the inner product of two p-dimensional vectors.

$$f(x) = \beta_0 + \sum_{i=1}^{n} \alpha_i \langle x, xi \rangle \tag{2.3}$$

Thus, the classifier is defined by parameters $\alpha_1, \ldots, \alpha_n$ and β_0, which are estimated by computing the inner product of all pairs of training observations. Once trained, classifying a new vector does not depend on computing inner products with all the training vectors, but only on those that lie in the margin or in the wrong side of the margin for their class, where $\alpha_i \neq 0$. These training vectors are known as support vectors.

SVMs extend the idea of support vector classifiers by replacing the inner product by a function K known as *kernel*. The simplest function K is the linear kernel, which reduces the SVM to a support vector classifier. However, different functions can be used to take features to a higher dimensional space, allowing for more complex decision boundaries. One of the most used kernel functions for SVMs is the Radial Basis Function (RBF), which is defined in Equation 2.4, where γ is a positive constant.

$$K(x^i, x^j) = \exp\Big(-\gamma \sum_{k=1}^{p} (x_k^i - x_k^j)^2 \Big) \tag{2.4}$$

The advantage of using kernels is that it is far easier to compute inner-products than to compute function values in higher dimensional spaces, which might even be infinite and therefore intractable. One feature that makes the RBF kernel so popular is the fact that it only incorporates one additional parameter (γ). Thus, when training a SVM with the RBF kernel, only two parameters need to be fine-tuned: C for the SVM and γ for the RBF kernel.

2.4.2.2 Random Forest

Random Forest (RF) is a tree-based machine learning method used for regression and classification tasks (James et al., 2014). The core elements in RF are decision trees. Decision trees involve segmenting the predictor space into a certain number of regions and can be applied to both regression and classification problems. Regions are defined as high-dimensional boxes and, once defined, the same prediction is made for every test observation that falls in a given region. In the case of classification trees, the predicted class for a given observation is given by the most commonly occurring class in the region in which the observation falls. Regions are usually defined following a recursive top-down greedy approach that begins with all observations as part of the same region and iteratively splits the predictor space. Each split is represented by two new branches down the tree. The approach applies a greedy heuristic decomposition since the best local split is made in each step. A usual split criterion in classification trees is the Gini index, defined by Equation 2.5, where K is the number of possible classes and $\hat{p}_{mk}$ represents the proportion of training observations that fall in region m and correspond to class k.

$$G = \sum_{k-1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) \tag{2.5}$$

The Gini index measures the variance of classes in a given region: a small value indicates that most observations in the region correspond to a single class. The iterative splitting strategy ends when all regions are comprised of observations corresponding to the same class or a certain stopping criteria is met (e.g., no region has more than a given number of training observations).

Decision trees are not very robust, since a small change in the training data can lead to a major change in the estimated tree. When used individually, decision trees usually show high variance and a tendency to overfit the

training set. RF aims to overcome these issues by incorporating two distinct features: aggregating many decision trees to improve their predictive accuracy and slightly modifying the split algorithm in decision trees during training.

Regarding the first feature, RF is an ensemble learning technique that builds multiple decision trees which are then combined to yield a single prediction. This is made possible by applying bootstrap aggregation to decision trees. Consider a classification problem for which a training set of observations $x_1, x_2, \ldots, x_n$ corresponding to classes $y_1, y_2, \ldots, y_n$ is given. RF repeatedly selects a random sample of observations (with replacement) from the training set and fits a decision learning tree after each draw. After training, predictions for an unseen observation are made by each trained tree and are aggregated to give the final prediction, usually using a voting mechanism. The voting mechanism can be as simple as a majority vote or can incorporate weights to the predictions of each tree based on the ratio of training observations of the same class in the predicted region.

The second distinct feature in RF is included during training. When computing the best possible split while building a decision tree, only a random subset of features is considered as a split candidate (typically, $\sqrt{p}$ features are sampled, where p is the total number of features). The rationale of this procedure is that, in the case that a feature is very strong, all the decision trees will start by using this feature as their top split. As a result, all the trees in the RF would be very similar. By restricting the possible features considered for each split, RF decorrelates individual decision trees, significantly improving their prediction accuracy.

2.4.3 Machine learning libraries

Three machine learning libraries were used for the implementation of VS, which are described next.

2.4.3.1 Scikit-learn

Scikit-learn is an open-source machine learning library for Python (Pedregosa et al., 2011). The library provides a collection of algorithms for both supervised and unsupervised learning, as well as tools for data processing, parameter tuning, and model evaluation. Scikit-learn was used in this thesis for the application of VS to the BSP. For that problem, RF was used as a classifier.

The scikit-learn implementation of RF uses the Gini index as a split criterion and combines classifiers by averaging their probabilistic prediction, instead of using a majority vote.

2.4.3.2 LIBSVM

Library for Support Vector Machines (LIBSVM) is a framework developed by Chang and Lin (2011) that supports vector classification, regression, and distribution estimation problems. It was designed to help users from outside the machine learning field to easily use SVM as a research tool. LIBSVM provides a simple interface for users to link with their programs. The main features of this framework include several SVM formulations, efficient multi-class classification, cross-validation, probability estimates, various kernels (e.g., linear, polynomial, RBF, and sigmoid), and weighted SVM for unbalanced data. It is implemented in both C++ and Java and has interfaces for many other languages.

2.4.3.3 xphi-LIBSVM

The original LIBSVM code does not directly provide support for parallelism. As part of this thesis xphi-LIBSVM was developed, which is a parallel version of LIBSVM specifically adapted to the latest Intel®Xeon Phi™ architecture (Massobrio et al., 2018c). The details of the implementation are outlined next.

A simple way to exploit the availability of multiple cores in a massively-parallel architecture is by using OpenMP to parallelize the loop that processes each training vector. This modification works with any C++ compiler that supports OpenMP. In order to exploit the specific characteristics of the Intel®Xeon Phi™ architecture, and to be able to use the Intel®MKL, the proposed xphi-LIBSVM uses the Intel®C++ compiler. The compiler options were set according to the recommendations of the Intel®MKL Link Line Advisor (Intel®Software, 2012) to tailor the compiled code to the specific hardware architecture.

An initial profiling was performed to identify bottleneck functions on the original LIBSVM code. The profiling was performed using *gprof* (Graham et al., 1982) and training on the *connect-4* dataset (Lichman, 2013) available at the LIBSVM dataset repository (Chang and Lin, 2011). The profiling revealed that 87.2% of the total training time was spent in the `dot` function of class

`Kernel`. This function performs the dot product of two vectors. To improve the overall training time the `dot` function was replaced by the threaded `cblas_ddot` routine, which is available in the BLAS Level 1 group of functions and routines of Intel®MKL.

Using the `cblas_ddot` function instead of the original code in LIBSVM is not as straightforward as interchanging only the calls to the functions, due to the format in which LIBSVM stores the training vectors. LIBSVM uses a "sparse" format, in which zero values are not stored. Instead, training vectors are stored as `<index:value>` pairs. For instance, the training vector `<0,1,0,3>` is internally represented as `(2:1 4:3)`. This format does not allow using the `cblas_ddot` function directly. Therefore, it was necessary to modify other sections of the LIBSVM code to implement a "dense" format, in which vectors are directly stored as arrays, including zero values. This design decision may achieve better or worse performance depending on the specific characteristics of the training set used. Therefore, xphi-LIBSVM users can decide at compiling time whether to use the original "sparse" format or the proposed "dense" format, depending on the specific characteristics of the training dataset.

The evaluation of the xphi-LIBSVM framework is presented next.

2.5 Experimental evaluation of xphi-LIBSVM

This section reports the experimental evaluation of the proposed xphi-LIBSVM framework.

2.5.1 Execution platform and problem instances

The xphi-LIBSVM framework was evaluated on an Intel®Xeon Phi™ 7250 processor, with 68 cores, and 64GB of RAM. The server was not shared with other users or performed any other intensive tasks during the experiments, in order to accurately measure the execution times.

Three learning datasets were used for the experimental evaluation of the proposed implementation. These datasets were obtained from the LIBSVM repository (Chang and Lin, 2011). The datasets, which correspond to classification and regression problems, are:

- *gisette*, a dataset corresponding to a handwritten digit recognition problem with the goal of separating the digits '4' and '9' (Guyon et al., 2004).

- *E2006*, a dataset with reports from thousands of publicly traded U.S. companies, published in 1996–2006, and stock return volatility measurements in the twelve-month period before and the twelve-month period after each report (Kogan et al., 2009).
- *usps*, a database for handwritten text recognition research, consisting of digitalized images at 300 pixels/in in 8-bit gray scale, corresponding to U.S. post codes scanned from real mail (Hull, 1994).

The datasets *gisette* and *E2006* were sub-sampled using a stratified selection of training samples, keeping the rate of appearance of each label in the dataset. A subset of 1000 samples of each dataset was used, which are hereinafter referred to as *gisette_1000* and *E2006_1000*. Table 2.1 shows the main characteristics of each of the datasets used in the experimental evaluation, including the type of problem (classification or regression), the number of training vectors (# samples), the length of each training vector (# features) and, for classification problems, the number of classes (# labels).

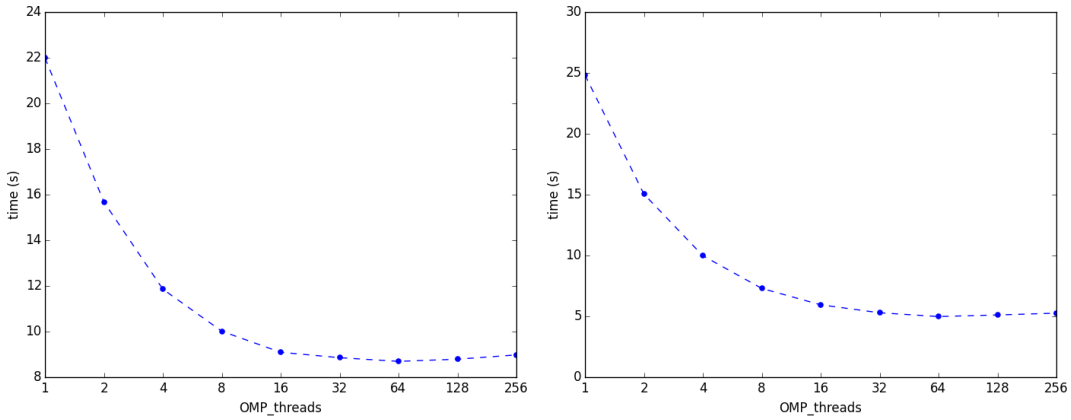
Table 2.1: Datasets used for the experimental evaluation of xphi-LIBSVM.

	problem	*# samples*	*# features*	*# labels*
gisette_1000	classification	1000	5000	2
E2006_1000	regression	1000	150 360	-
usps	classification	7291	256	10

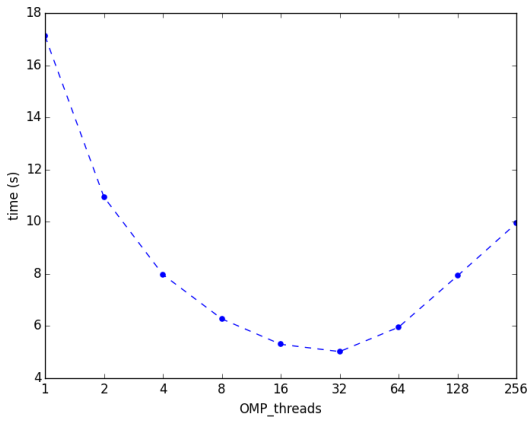
2.5.2 Coarse-grain parallelization

Initially, the experimental evaluation focused on the coarse-grain parallelism of the outer loop that performs the kernel evaluations. These results correspond only to the outer loop parallelization, without changing the vector representation format and without using Intel®MKL. Figure 2.6 shows the average execution time in seconds for the three studied instances, when varying the number of threads (set by the `OMP_NUM_THREADS` environment variable) assigned to the outer loop. The results correspond to 30 independent executions of each instance with each studied number of threads.

Results in Figure 2.6 show that acceptable execution time improvements were achieved when using more than one core on all studied instances. However, execution times did not improve when using more than 64 cores for both



(a) *gisette_1000* **(b)** *E2006_1000*



(c) *usps*

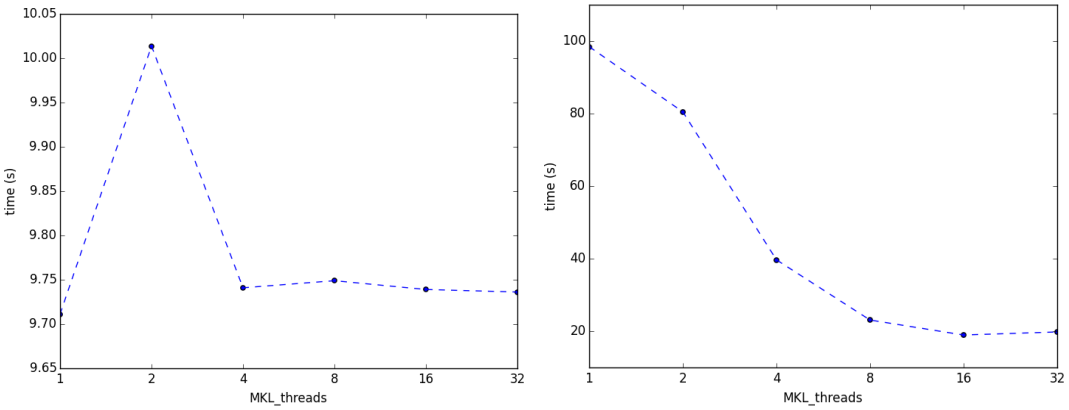
Figure 2.6: Mean execution time of xphi-LIBSVM with different number of OMP threads.

gisette_1000 and *E2006_1000* datasets, and there was even a significant negative impact when using large number of cores with the *usps* dataset. This could be explained due to the fact that the Intel®Xeon Phi™ processor used has 68 physical cores. Therefore, when using more threads, some of the CPU resources are shared among the threads, incurring in a noticeable overhead.

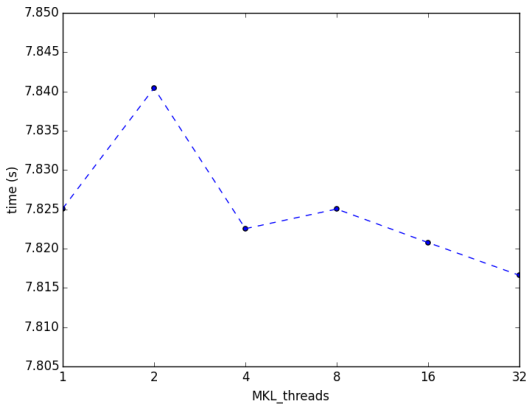
2.5.3 Vectorized dot product computation

Afterward, the experimental analysis studied the performance when changing from a "sparse" to a "dense" vector representation and including the Intel®MKL routines for the dot product calculation. Figure 2.7 shows the average execution time in seconds for the three studied instances, when varying the number of threads (set by the `MKL_NUM_THREADS` environment variable)

assigned to the dot product calculation. There is no coarse-grain parallelization in these executions (i.e., `OMP_NUM_THREADS` = 1). The results correspond to 30 independent executions of each instance with each studied number of threads.



(a) *gisette_1000* **(b)** *E2006_1000*



(c) *usps*

Figure 2.7: Mean execution time of xphi-LIBSVM with different number of MKL threads.

Results in Figure 2.7 give information on two aspects. Firstly, on the convenience (or not) of using the "dense" format and including Intel®MKL for the dot product calculation. Secondly, to discuss the usefulness of adding parallelism at the vector level when computing the dot product.

To discuss the first aspect, execution times when using only one OMP thread in Figure 2.6 should be compared against those achieved when using one MKL thread in Figure 2.7. It can be observed that execution times significantly improved when running a sequential version with the "dense" representation format and the Intel®MKL dot product calculation for both

gisette_1000 and *usps* instances. However, for *E2006_1000* instance, since the training vectors are much larger, using the "dense" format and only one MKL thread negatively impacted the execution time. In this case, the effects of the "dense" representation were only mitigated when adding more MKL threads to reduce the execution times.

Regarding the second aspect, results showed that when using the "dense" format, the improvements achieved by using a larger number of threads for the dot product computation were only noticeable for very large vectors. For *usps* instance, with vectors of size 256, the improvements when using more than one thread were marginal. For *gisette_1000*, with vectors of size 5000, there was even some minor performance decline when using more than one thread. Additionally, there was a strange behavior when using exactly two threads, possibly due to the overhead of creating the pool of threads. However, for instance *E2006_1000*, with training vectors of size 150 360, there was a noticeable improvement when using more threads for the dot product calculation.

In conclusion, dense vectors benefit from changing the original LIBSVM representation and using Intel®MKL for the dot product calculation, but only dense and large vectors benefit from using multiple threads when computing each dot product. The proposed implementation allows the user to control both the vector representation and the number of outer (OMP) and inner (MKL) threads, thus, enabling the user to tune the library to the specific needs of their learning task.

2.5.4 Two-level parallelization approach

Taking into account the results discussed in the previous sections, thirty independent executions of each training dataset were performed using the configuration of threads that achieved the best results. The selected configurations are reported in Table 2.2, where `OMP_NUM_THREADS` indicates the number of threads used for the outer loop of kernel evaluations (i.e., coarse-grain parallelism) and `MKL_NUM_THREADS` indicates the number of threads assigned to compute each vector dot product, when the "dense" format is used. Additionally, thirty independent executions of the original LIBSVM library were performed over each dataset.

Table 2.2: Thread configurations used for each problem instance.

	format	OMP_NUM_THREADS	MKL_NUM_THREADS
gisette_1000	dense	64	1
E2006_1000	sparse	64	-
usps	dense	32	32

Table 2.3 presents the execution times achieved by the original LIBSVM code and the proposed xphi-LIBSVM using the best configuration for each problem instance. For each instance the minimum (best), average, and standard deviation of execution times are presented with the following format: mean±std (min). All times are expressed in seconds. Additionally, the average acceleration achieved is presented for each instance, which is computed as the ratio between the average execution time of LIBSVM and the average execution time of the proposed xphi-LIBSVM implementation.

Table 2.3: Execution time in seconds (mean ± std (min)) and average acceleration of xphi-LIBSVM vs. LIBSVM.

	LIBSVM	*xphi*-LIBSVM	$\overline{\text{acceleration}}$
gisette_1000	22.66 ± 0.06 (22.59)	7.07 ± 0.02 (7.03)	3.21x
E2006_1000	20.59 ± 0.03 (20.56)	4.98 ± 0.02 (4.96)	4.13x
usps	18.46 ± 0.06 (18.24)	3.84 ± 0.01 (3.81)	4.81x

Results in Table 2.3 show that the proposed implementation can efficiently improve the training time while computing the same results than the original LIBSVM. The proposed implementation achieved an acceleration of up to 4.81x on average on the *usps* dataset. These training time improvements are significant, especially when considering larger training datasets that would otherwise be intractable for sequential SVM implementations. The best results were achieved for classification problems with large and "dense" training vectors, which take the most advantage of the vectorized dot product calculation provided by Intel®MKL.

Chapter 3

Related Work

This chapter outlines the review of works in the literature related to VS. Firstly, an overview of works related to the automatic generation of parallel programs is presented in Section 3.1. Then, Section 3.2 reviews works that deal with the application of machine learning to solve optimization problems. Methods inspired in the Savant Syndrome are presented in Section 3.3. Finally, Section 3.4 summarizes the findings of the review of related works and reflects on the contribution of this thesis to the existing literature.

Works related to the optimization problems used in this thesis for the evaluation of VS are not presented in this chapter but in the chapters corresponding to each specific problem addressed.

3.1 Automatic generation of parallel programs

VS aims to automatically generate—via learning—a solver that can be run in parallel for a given optimization problem. The automatic generation of parallel programs has been addressed in the past.

One research line in the literature corresponds to the parallelization of existing sequential programs by applying source-to-source transformations, which rely on carefully inspecting dependencies to identify possible segments in the original program that can be parallelized.

An early proposal for automatic parallelization of sequential code was presented by Irigoin et al. (1991). In their work, the authors proposed a source-to-source parallelizer implemented in Fortran. The parallelizer performs a

semantical analysis of the original source code and makes a series of transformations including: privatization of variables, to remove dependencies; loop distribution, where a loop with multiple (independent) instructions is replaced by a set of loops that can be run in parallel; and nested loop parallelization, which interchanges the order in nested loops in order to maximize the efficiency when accessing memory. The proposed framework also supports computing platforms with dedicated vector processing units.

The work of Midkiff (2012) describes the concepts behind the automatic parallelization of unaltered and unannotated sequential programs. The author outlined the fundamental principles used by compilers to parallelize numerical programs, including source code analysis strategies to identify interactions and dependencies, and code transformations aimed to expose parallelism targeted at multicore and vector processors. The author focused on shared-memory systems but also discussed parallelizing programs for execution on distributed-memory platforms.

Some works in the literature have applied Genetic Programming (GP) (Koza, 1992) to achieve automatic parallelization.

Ryan and Ivan (2000) presented Paragen, an automatic parallelization system that combines GP and genetic algorithms (GAs). Paragen generates parallel programs that are functionally-equivalent to a sequential program given as a reference. The source code of the sequential program is used as a starting point and several transformations are iteratively applied until the framework produces a parallel version of the program. GP is used to apply atomic transformations (i.e., those that affect a single instruction) while a GA is used for transformations involving loops. The method was evaluated on a very trivial program involving loops and simple arithmetic operators and showed good preliminary results. The authors highlighted the importance of increasing the number of possible transformations and applying Paragen to solve more complex benchmark problems.

Cheang et al. (2006) proposed using GP to generate entirely new programs that can be run in parallel. The proposed paradigm was devised for Multi-Arithmetic-Logic-Unit Processors, a type of tightly-coupled register machines. The goal of the authors was to create parallel programs specifically adapted to this underlying architecture without human intervention. They used GP to evolve sequences of parallel-instructions selected from a defined set. The implementation was evaluated over a benchmark of fourteen problems, includ-

ing numeric and Boolean functions, Fibonacci sequence computation, among others. Experimental results showed that it was easier to evolve parallel programs than sequential counterparts. The authors concluded that their approach needed to be extended with application-specific functions to solve more complex problems.

Few works have applied machine learning to automatically generate programs that can run in parallel.

Tournavitis et al. (2009) argued that traditional static parallelism detection techniques are ineffective due to the lack of information in the static source code. Additionally, the authors highlighted that previous approaches did not take into account the mapping of the parallel program to the underlying architecture. To overcome these issues, the authors proposed a framework for automatic parallelization. The proposed framework is comprised of a profile-driven parallelism detection phase and a mapping mechanism based on machine learning. Starting from a sequential code, the framework performs a dependence analysis that builds upon traditional static analysis by incorporating profiling information. The result is a parallel code with OpenMP annotations that are later extended with allocation clauses using a machine learning model trained with a set of known parallelization strategies. However, the framework needs user intervention to approve the generated code since correctness is not guaranteed.

3.2 Learning for optimization

The idea of applying machine learning techniques to solve optimization problems has been addressed in the literature. A review of works in this area of study is presented next.

Many real-world optimization problems follow the predict-then-optimize paradigm, which consists of sequentially predicting a set of unknown parameters or variables of the problem instance (e.g., travel times in a Vehicle Routing Problem (VRP)) and then using an optimization solver to provide a solution (e.g., near-optimal routes in the VRP) (Grimes et al., 2014; Demirović et al., 2019).

Elmachtoub and Grigas (2017)[1] proposed Smart "Predict, then Optimize" (SPO), a framework for problems that follow the predict-then-optimize paradigm. The authors argued that traditional approaches for these problems do not take into account how predictions in the first stage affect the optimization in the second stage. Instead of focusing on prediction errors, the authors proposed the SPO loss function, which measures the decision error induced by a wrong prediction. Thus, the proposed SPO framework trains a machine learning model that learns with respect to the error in the optimization problem instead of the error in the learning task. Due to the intractability of the SPO loss function, the authors proposed a surrogate loss function that can be computed using stochastic gradient descent. Experimental evaluation on synthetic instances of the shortest path and portfolio optimization problems showed that the proposed framework was able to outperform classic predict-then-optimize approaches.

Later, Mandi et al. (2020) extended the work of Elmachtoub and Grigas (2017) to solve more realistic discrete optimization problems. The main challenge in the original SPO framework is the need for repeatedly solving the optimization problem during learning. Consequently, the authors proposed strategies to relax the problem and offered ways to warm-start the learning process using previous solutions. Experimental evaluation was performed on small instances of the weighted knapsack and scheduling problems. Additionally, experiments on five hard scheduling instances were performed, showing that the proposed approach outperformed a traditional two-stage approach that did not considered an optimization-directed loss.

Another line of work in the related literature consists of introducing combinatorial building blocks within machine learning algorithms.

Vlastelica et al. (2020) proposed an end-to-end architecture that integrates blackbox implementations of combinatorial solvers into neural networks. The proposed architecture can be used along with any combinatorial solver that optimizes a linear function. The main idea behind the method consists in applying a novel interpolation technique that allows computing gradients of the piecewise constant function that characterizes the combinatorial optimization problem. The authors trained architectures with implementations of Gurobi (a general-purpose mixed integer linear programming solver), Dijkstra's shortest-path algorithm, and a state-of-the-art implementation of a minimum

[1]Unpublished work (arXiv preprint arXiv:1710.08005)

cost perfect matching algorithm. Experimental evaluation was performed over three optimization problems: shortest path finding, Traveling Salesman Problem (TSP), and minimum cost perfect matching. The proposed architecture showed high accuracy and generalization capabilities. The authors proposed embedding approximate solvers as future work, to be able to address real-world problem instances that are not tractable using exact solvers.

Following a similar strategy, Berthet et al. (2020) proposed a systematic method that transforms optimizers into differentiable operations that can be learned and embedded into machine learning pipelines. The approach relies on stochastically perturbing discrete optimizers with random noise and considering the perturbed solutions to the problem. The proposed strategy was evaluated on the same shortest path problem instances used by Vlastelica et al. (2020), outperforming their results in terms of accuracy and cost ratio.

Many works in the literature have focused on learning optimization solvers for problems that are modeled using graphs, for which specific architectures have been proposed.

Vinyals et al. (2015) introduced Pointer Networks (PNs), a model based on Recurrent Neural Networks (RNNs). The proposal aimed to solve a common shortcoming of previous approaches that used RNNs to learn functions over sequences: the size of the output dictionary needs to be fixed and known beforehand. For this purpose, the authors proposed PNs, where an encoder is used to parse an input graph and produce an encoding for each node in the graph. Then, a decoder produces a probability distribution over these nodes, following an attention mechanism that resembles the one proposed by Bahdanau et al. (2015). By repeating this decoding step, the PN is able to output a permutation of the nodes of the graph given as input. Thanks to its design, PNs can handle graphs of arbitrary size. The proposed model was evaluated when solving three discrete combinatorial optimization problems: finding planar convex hulls, computing Delaunay triangulations, and solving the planar TSP. PNs were trained by observing solved instances of each problem and, similarly to the VS paradigm, can also deal with problem instances of varying size. Experimental results showed that PNs were able to find competitive results in problem instances larger than those seen during the training phase.

Later, Bello et al. (2017) outperformed the results of Vinyals et al. (2015) when solving the TSP by using reinforcement learning over the PNs architecture, using the inverse of the tour length as a reward signal. By using rein-

forcement learning, the method does not depend on the availability of solved instances for the problem. Experimental evaluation was performed on TSP instances of up to 100 cities. To outline the applicability of the proposed approach to other optimization problems, the authors also studied the 0/1-KP. For this problem, the experimental evaluation was performed on instances of up to 200 items. The proposed approach found solutions to the studied instances that were within 1% of the known optima. The authors concluded that tackling problems with harder constraints is a difficult task and proposed incorporating a penalization scheme into the reward function as a line of future work. In Chapter 4, the VS framework is evaluated when solving the NRP, a software-engineering optimization problem that is modeled as a 0/1-KP, on instances of up to 1500 items.

More recently, Hu et al. (2017) extended the model proposed by Bello et al. (2017) and applied it to the three-dimensional bin packing optimization problem. The goal of the problem is to design a bin that can pack all the items in the instance while minimizing the surface area of the bin. A deep reinforcement learning approach was used to predict the sequence in which items are packed. A PN receives as input the elements that need to be packed and outputs a permutation of these items that indicates the order in which the items should be added to the bin. The PN was trained using reinforcement learning, according to the surface of the smallest bin that can pack all the items in the output sequence. The specific empty spaces in which the items are placed and the orientation of each item were computed separately, using heuristic methods. The proposed approach outperformed a specific heuristic for the problem during the experimental analysis. Improvements of 5% on average over the baseline heuristic were achieved for the studied instances.

Kool et al. (2019) proposed an attention model trained with reinforcement learning. The proposed model receives a graph as an input and sequentially chooses nodes to add to a tour until a full tour has been constructed. The model is independent of the order in which the nodes are given as input. The authors trained their model using REINFORCE, a policy-gradient based algorithm. The authors showed the flexibility of their approach over the TSP, two variants of the VRP, and other routing problems over graphs. Experimental results showed that the proposed model significantly improved over learned heuristics on problem instances with up to 100 nodes. The authors concluded that larger instances should be addressed in the future.

Khalil et al. (2017) proposed using a single model based on graph embeddings instead of a separate encoder and decoder as used by Vinyals et al. (2015). The authors advocated for taking advantage of the fact that, when solving optimization problems, the structure tends to remain nearly unchanged, with problem instances only varying in the specific data. Thus, they proposed a framework that combines reinforcement learning with graph embedding to solve optimization problems over graphs. The proposed approach greedily constructs solutions and takes advantage of a graph embedding network called *structure2vec* to incorporate the graph structure into the learning process. Three optimization problems over weighted graphs were used for the experimental evaluation: the Minimum Vertex Cover, the Maximum Cut, and the TSP. Experimental results showed that the proposed approach outperformed traditional heuristics for the problem, both in synthetic and real-world problem instances. Results were better for problems where the graph structure is important to compute the overall solution.

Other combinatorial optimization problems—besides those modeled through graphs—have also been addressed.

Selsam et al. (2019) proposed *NeuroSAT*, a solver for the propositional satisfiability problem (SAT) based on Message Passing Neural Networks (MPNN). The proposed approach relies on training a MPNN using only the satisfiability of the problem instance as a supervision bit. Experimental evaluation showed that the network was able to predict satisfiability after several iterations. A posthoc procedure based on clustering was used to derive the Boolean values of each variable based on the activations in the neural network. The experimental evaluation showed that NeuroSAT was able to solve larger instances than those used during training, albeit demanding a larger number of iterations and incurring in a significant drop in accuracy. No execution time or performance metrics were reported either for the training or prediction experiments.

Two recent works have applied reinforcement learning to solve scheduling problems.

Waschneck et al. (2018) applied Deep Q Network (DQN) to the Job Shop Scheduling Problem in a factory environment. Cooperative DQN agents were trained using reinforcement learning according to user-defined objectives related to optimization in production scheduling. Each agent optimized the rules corresponding to one workcenter in the factory and monitored the actions of other agents to optimize a global reward. Experimental analysis over a small

factory simulation showed that the proposed approach achieved solutions of comparable quality to those computed by an expert but was not able to outperform classic dispatching heuristics.

More recently, Wang et al. (2019) applied a similar approach to that of Waschneck et al. (2018) to study the multi-objective workflow scheduling problem. A DQN reinforcement learning model was used to schedule computing tasks in a cloud environment with the goal of minimizing task completion time and the cost for the user. The experimental evaluation of the model was done using well-known scientific workflow templates as well as real data from the Amazon EC2 cloud. The experimental evaluation showed that the proposed approach was able to outperform several baseline heuristics and metaheuristics for the problem.

Finally, a recent paper by Bengio et al. (2021) surveyed the literature on machine learning for combinatorial optimization, outlined a methodology for further integrating both fields of research, and enumerated the main challenges in this regard. Based on their survey, the authors concluded that "Although most of the approaches we discussed in this paper are still at an exploratory level of deployment, ... we strongly believe that this is just the beginning of a new era for combinatorial optimization algorithms".

3.3 Savant-inspired computational methods

According to the literature reviewed, the first attempt to mimic the behavior of savants in a computational model was presented by Norris (1990). In this work, the author presented a model of a calendrical savant using ANNs. The proposed model aimed at predicting the day of the week in which a given date falls, using a set of dates labeled with its corresponding day of the week as training data. Initially, a simple single-layered neural network was devised to predict dates between 1950–1999. The neural network had 31 inputs to code the day, 12 inputs to code the month, and 15 inputs to code the year (which was split between the decade and the year within the decade). The output of the neural network consisted of seven units, one for each day in the week. Training of the network was done using back-propagation on a set of randomly-selected dates corresponding to one fifth of the considered period. When predicting dates not seen during training, the neural network exhibited poor performance, slightly above chance, showing poor generalization. The

author argued that savants probably do not learn from examples taken at random, but rather learn first about days, then months, and finally years. To mimic this strategy, the author proposed a cascade learning model comprised of three subnetworks: one focused on learning about days, one in months, and one in years. Each subnetwork was iteratively trained. The first subnetwork, which receives the day of the month as input, was trained with all dates from January 1950. Once trained, the second subnetwork, which receives the month and the output from the first subnetwork as input, was trained using every date in 1950. Finally, the third subnetwork, which receives the year and the output from the second subnetwork as input, was trained using one fifth of randomly-selected dates from the whole period. With this design, the network achieved an accuracy of 90% when predicting dates not seen during training, which is comparable to the best calendrical savants reported in the literature. Most errors produced by the network corresponded to dates in leap years, with up to four times as many errors when compared to non-leap years. The author concluded that, although the model was able to perform accurately, the learning scheme was not completely automated since it used problem-specific knowledge (i.e., training separately for days, months, and years). To overcome this issue, the author posed that multi-layer networks may be a suitable solution for the task. However, training multi-layer networks involved a significant computational effort for the hardware available at the time this paper was published.

Weijters (1995) extended Norris's approach to learn how to predict the day of a given date by combining neural networks with Self Organizing Maps (SOMs). The author proposed an architecture that combines multi-layered feed-forward neural networks with SOMs (as many as hidden layers in the network). While the neural network is trained, the SOMs are also trained using the hidden-unit activations of the neural network as input. Then, information from the SOMs is used when updating the connection weights in the neural network. Weijters highlighted Norris's remarks stating that his proposed neural network would not be able to solve the calendar calculation task had it not had human assistance. In his experimental analysis, Weijters replicated the experiments performed by Norris, using the same date ranges and a similar network topology. The proposed approach of neural networks combined with SOMs was compared against the simpler neural network with backpropagation proposed by Norris. Results showed a great improvement of

the proposed approach over the original model presented by Norris, achieving an average classification error below 3%. Later, Weijters et al. (2000) further demonstrated that a single hidden-layer feedforward neural network combined with a SOM can become a successful date calculator, thus challenging Norris's statement that expert-based knowledge is mandatory to solve the problem.

The first version of VS was presented by Pinel et al. (2013). In this work, an independent task scheduling problem was addressed, which is further studied in this thesis in Chapter 5. The authors presented the VS paradigm, its analogy to the Savant Syndrome, and described the model as a single iteration of a MapReduce application. The optimization problem addressed consists in assigning independent tasks to computational resources such that the finishing time of the last task is minimized. Multi-class SVMs were used to predict the machine assigned to a given task and a random LS was used as an improvement operator to refine the predicted assignment. The training of the model was done using solutions computed by MinMin, a well-known greedy heuristic for the problem. The experimental evaluation was done using 100 instances of 128×4 and 512×16 (i.e., tasks×machines). Considering only the prediction phase, VS was able to compute solutions 10% worse (in median) than MinMin in the small instances and even outperform MinMin after the application of the LS. In the case of the largest instances, the improvement step was necessary to compute comparable solutions to those of the reference algorithm. Later, Pinel and Dorronsoro (2014) provided a more in-depth description of the model, additional details of the feature selection strategy, and an extended experimental evaluation where better results were computed thanks to an increase in the number of training observations.

More recently, Dorronsoro and Pinel (2017) addressed the same task scheduling problem but proposed a different operator for the improvement phase of VS. In this work, the assignment probabilities returned from the SVMs were used to generate the initial population of a Parallel Asynchronous Cellular GA previously designed for the problem. Two variants of VS were presented: one that was trained based on the solution computed by MinMin and one that used the genetic algorithm as a reference. The experimental evaluation showed that both variants were able to improve the results of the reference genetic algorithm (which used the MinMin solution as a seed). Particularly, in the case of small instances (i.e., 12×4 and 128×4), VS was able to find better solutions in half a second than those found by the reference algorithm in 10

seconds. Experiments demonstrated that initializing the population with the predictions from the SVMs helped the GA to start from better regions of the search space and to compute accurate solutions faster. An extended experimental evaluation (including a comparison to state-of-the-art solvers for the problem) along with further details on the training process of VS was later presented by Pinel et al. (2018).

In Chapter 5 this task scheduling problem is further addressed. The main contributions over the previous works include studying larger problem instances (up to 65536×16) and assessing the scalability of VS when using varying number of resources in different computing platforms, including multi-core and many-core systems, and a cluster of distributed computing nodes.

3.4 Summary and discussion

Table 3.1 summarizes the related works included in the literature review, providing a brief comment on each reviewed work.

The automatic generation of parallel programs has already been studied in the literature, with initial proposals dating back as far as thirty years. VS, the paradigm explored in this thesis, is related to this line of research, since it generates solvers for optimization problems that can run in parallel. Some of the reviewed works applied source-to-source transformations while others used GP to evolve a program in order to achieve parallelism. These approaches to parallelization preserve the algorithm and most of the source code, since they apply transformations that respect the semantics of the original program. In contrast, VS does not need access to the source code of the algorithm used as a reference. Moreover, VS can even use a set of previously-solved instances of the problem being addressed. Another limitation of the reviewed works is that the programs used for evaluation are extremely simple (in terms of the number of instructions). In this regard, VS is evaluated when solving large instances of three complex optimization problems.

The analysis of related works shows an increasing interest in applying machine learning to solve optimization problems. This interest is very recent, with most of the reviewed works published within the past five years. Many of these works aimed to learn solvers for optimization problems defined over graphs. However, other optimization problems were also addressed in the literature, including variants of some of the problems studied in this thesis.

Table 3.1: Summary of the related works included in the literature review.

Automatic generation of parallel programs	
reference	*comment*
Irigoin et al. (1991)	Proposed an automatic source-to-source parallelizer.
Midkiff (2012)	Addressed the automatic parallelization of unaltered and unannotated sequential programs.
Koza (1992)	Proposed GP, which some authors used to automatically generate parallel programs.
Ryan and Ivan (2000)	Proposed Paragen, an automatic parallelization system based on GP and GAs.
Cheang et al. (2006)	Used GP to build parallel programs for a tightly-coupled computing architecture.
Tournavitis et al. (2009)	Proposed a framework for automatic parallelization considering the underlying architecture.
Learning for optimization	
reference	*comment*
Grimes et al. (2014) Demirović et al. (2019)	Used a predict-then-optimize approach to solve optimization problems.
Elmachtoub and Grigas (2017)	Introduced SPO, a framework for problems that follow the predict-then-optimize paradigm.
Mandi et al. (2020)	Extended the work of Elmachtoub and Grigas (2017) with relaxations for discrete problems.
Vlastelica et al. (2020)	Integrated blackbox implementations of combinatorial solvers into neural networks.
Berthet et al. (2020)	Integrated perturbed optimizers into learning pipelines.
Vinyals et al. (2015)	Introduced PNs based on RNNs and solved discrete combinatorial problems.
Bahdanau et al. (2015)	Proposed the attention mechanism used in PNs.
Bello et al. (2017)	Combined PNs with reinforcement learning to solve the TSP and 0/1-KP.
Hu et al. (2017)	Combined PNs with deep reinforcement learning to solve the three-dimensional bin packing problem.
Kool et al. (2019)	Proposed an attention model with reinforcement learning for routing problems over graphs.
Khalil et al. (2017)	Combined reinforcement learning with graph-embedding to solve graph problems.
Selsam et al. (2019)	Proposed a model based on MPNN for SAT.
Waschneck et al. (2018) Wang et al. (2019)	Applied DQN to solve single and multiobjective scheduling problems.
Bengio et al. (2021)	Reviewed the application of machine learning to combinatorial optimization.
Savant-inspired computational methods	
reference	*comment*
Norris (1990)	Proposed the first attempt to mimic the behavior of savants in a computational model.
Weijters (1995) Weijters et al. (2000)	Extended the work of Norris (1990) by combining neural networks with SOMs.
Pinel et al. (2013) Pinel and Dorronsoro (2014)	Introduced the VS model to solve an independent task scheduling problem.
Dorronsoro and Pinel (2017) Pinel et al. (2018)	Combined VS with a cellular GA to solve an independent task scheduling problem.

Some of the reviewed techniques involve iteratively executing the solver that is being learned. This constitutes a major obstacle when addressing large instances of complex optimization problems, since the cumulative time executing the solver during learning may be unbearable. Once again, the fact that VS can learn from a benchmark of previously-solved instances is a major advantage over these methods.

Many approaches in the literature only learn from exact solvers for the problem. This comes at a large expense to applicability, since these methods can only address problem instances up to the largest size that the exact solver can adequately handle. In this regard, the design of VS presents a major advantage over these approaches, since it can learn from instances solved by both exact and approximate algorithms. Out of the problems addressed in this thesis, one of them used solutions computed by an exact algorithm as reference (NRP) and the remaining two used approximate solutions as reference (HCSP and BSP). The experimental evaluation of VS when solving the HCSP and the BSP involved studying the scalability when solving problem instances that were significantly larger than those seen during training. This represents a major advantage over some of the reviewed works in the literature that can only scale up to a problem size that is tractable for the exact solver used as a reference.

Since many of the proposals in the literature correspond to novel techniques, experimental evaluation is sometimes not sufficiently exhaustive, with many approaches being evaluated only over synthetic and very small problem instances. This calls into question the applicability of such proposals to large real-world problem instances. In this thesis, VS is evaluated over three problems of vastly different fields. Out of these problems, the BSP addressed in Chapter 6, is a complex optimization problem in public transportation networks and VS is evaluated when solving realistic instances of this problem.

The computational performance of the proposed models found in the reviewed works is rarely studied and few mentions are made to the use of multiple computing resources in parallel. In contrast, Chapter 5 outlines the evaluation of the performance of VS over four different computing platforms. For each platform, the scalability of VS in terms of the use of computational resources is studied.

According to the literature reviewed, the first savant-inspired method was proposed by Norris (1990) and later extended by Weijters (1995); Weijters et al.

(2000). Since then, no other works in this regard were found in the literature up to the proposal of VS by Pinel et al. (2013). This thesis significantly expands the previous studies of VS by defining and implementing solutions to three highly-relevant optimization problems arising in different fields of study and evaluating its performance in different computational platforms.

Chapter 4

Virtual Savant for the Next Release Problem

This chapter presents how VS can be used to automatically learn from an exact algorithm how to solve the NRP. Section 4.1 presents the NRP formulated as a specific variant of the 0/1-KP and briefly reviews the related literature on the problem. Then, Section 4.2 outlines how the VS framework is adapted to solve the NRP and Section 4.3 presents the implementation details. The experimental evaluation is presented and discussed in Section 4.4, followed by some concluding remarks in Section 4.5.

4.1 The NRP and the 0/1-KP

This section introduces the NRP formulated as a 0/1-KP and presents a brief review on the literature related to both optimization problems.

4.1.1 NRP overview

Requirements Engineering (RE) can be described as the process of formulating, documenting, and maintaining a set of requirements during an engineering design process (Nuseibeh and Easterbrook, 2000). RE is an important discipline in many areas of engineering, and it is especially useful in software engineering, where defining and analyzing software requirements is crucial to properly define a system (Aurum and Wohlin, 2005). Particularly in early stages, or during the inception of the new release of a product, requirements can be classified into two groups. One group comprised of core or mandatory

requirements that cannot be dispensed with and will likely be present in any version of a product for technical, strategical, or policy compliance reasons. A second group, composed of optional requirements, usually reflecting different product features, which may be demanded by stakeholders. The selection of the optional requirements to implement is a key problem in RE.

The NRP is a related problem in Software Engineering (Bagnall et al., 2001). In essence, the NRP consists of selecting a subset of requirements or features to include in the next release of a software product, taking into account their expected revenues and other factors, such as which requirements are demanded by the stakeholders (users or customers) and their perceived relevance. This selection problem is constrained in practice by the resources available for the development process, as requirements have implementation costs that cannot exceed a given budget. There are also potential interactions between requirements, like dependencies or precedence constraints. The NRP is a relevant problem when it comes to developing and maintaining modern complex software systems and is one of the most popular problems for which search-based RE approaches have been applied.

Among the different NRP variants proposed in the seminal work by Bagnall et al. (2001), the basic independent NRP considers no dependencies between requirements, disjoint sets of requirements for different stakeholders, and a single objective function to maximize. The objective function takes into account the revenues of the requirements and the preferences of the stakeholders for the next release of a software product. The goal is maximizing the total revenue without incurring in a total cost that exceeds the available budget.

More elaborate NRP variants exist in the literature. For instance, dependencies among requirements are considered in the original general NRP formulation (Bagnall et al., 2001). However, many algorithms addressing these variants work by transforming instances into one or several instances of the basic independent NRP, thus effectively resorting to this simpler version to solve them under the hood. Consequently, the basic independent NRP is relevant even when considering more complex scenarios.

The basic independent NRP is the specific variant addressed in this Chapter and, for simplicity, it is referred hereinafter as NRP. The mathematical formulation of the problem is presented next.

4.1.2 NRP formulation

The NRP deals with the selection of a subset of requirements or features to include in the next release of a particular software product. Each requirement has an associated implementation cost and expected revenue, usually related to the stakeholders' preferences. The goal is to find the subset of requirements that maximizes the total revenue, subject to a budget limit for the total cost of the requirements included in the next software release.

At a lower level of abstraction, the NRP can be characterized as a specific variant of the 0/1-KP, a classical $\mathcal{NP}$-hard combinatorial optimization problem (Kellerer et al., 2004; Bagnall et al., 2001). This problem can be mathematically formulated as follows. Given a set of n items, each with a weight w_k and a profit p_k, the 0/1-KP consists in finding a subset of items to include in the knapsack that maximizes the total profit, without exceeding the weight capacity C of the knapsack. Equation 4.1 outlines the problem formulation, where decision variables $x_k \in \{0, 1\}$ indicate whether the corresponding item is included (1) or not (0) in the knapsack. The knapsack capacity is analogous to the budget in the NRP formulation while items model the possible requirements to include in the next software release, each with an associated cost (i.e., the item's weight) and a given revenue (i.e., the item's profit).

$$\arg\max \left\{ \sum_{k=1}^{n} p_k x_k \;\middle|\; \sum_{k=1}^{n} w_k x_k \leqslant C \right\} \tag{4.1}$$

4.1.3 Related work

The 0/1-KP is a widely studied problem in Operations Research. Dantzig (1957) studied this problem and called it "the knapsack problem", recognizing its importance from an integer programming perspective. Nemhauser and Ullmann (1969) proposed an exact algorithm to solve the 0/1-KP using dynamic programming. The algorithm was applied to solve a well-known problem in the field of management: allocating capital within constrained budgets. The basic capital allocation problem can be modeled as a 0/1-KP considering investment opportunities, or projects, as the items to select for inclusion in the knapsack and the available budget for investments as the knapsack capacity. Each project has a required investment and an expected return of investment, which are analogous to the weight and profit of items in the 0/1-KP formu-

lation. Thus, the capital allocation problem consists in finding a portfolio of projects in which to invest within a given constrained budget. A myriad of algorithms and techniques have been developed for the 0/1-KP. The books by Martello and Toth (1990) and, more recently, Kellerer et al. (2004) are authoritative sources on knapsack-related problems and algorithms, including both the underlying theory and a discussion on practical aspects.

Bagnall et al. (2001) introduced the NRP as an optimization problem of industrial importance in RE. Several variants of the problem are discussed in this work, where the objective function aims to maximize the satisfaction of stakeholders subject to different constraints. Given the limitations of exact algorithms, approximation algorithms and heuristics were used to solve the different variants.

Harman et al. (2014) studied the NRP modeled as a 0/1-KP and presented a sensitivity analysis tool to help stakeholders deal with inaccuracy when estimating the cost and revenue of requirements in a project. The underlying idea is to identify those requirements for which a small deviation in cost or revenue estimation leads to a large difference in the optimal requirements set. Once those problematic requirements are identified, a decision-maker can potentially assign more resources to the estimation process of these sensitive requirements. An optimized implementation of the original algorithm from Nemhauser and Ullmann (1969) was used to solve NRP instances modeled as a 0/1-KP. This optimized implementation is the one used as a reference algorithm during the training phase of VS.

Veerapen et al. (2015) solved different versions of the NRP, including single and bi-objective problems, using integer linear programming. CPLEX was used to solve the problem instances and it was observed that the performance of this approach significantly improved since Bagnall's seminal work. This improvement does not just stem from the dramatic increase in computing power, but also from the steady advances in integer linear programming solvers. In the approach proposed by Veerapen et al., the goal was to provide an exact optimization method capable of managing instances of reasonable size. Since the problem is $\mathcal{NP}$-hard, execution times grow dramatically for large problem instances.

Some works in the literature have used machine learning to address the NRP. Araújo and Paixão (2014) proposed an architecture that combined machine learning with an interactive genetic algorithm to solve the NRP. In their

proposed approach, a machine learning model was used to replace the human interaction needed to model the requirements engineer preferences. The goal was to eventually replace the human interactions needed for the execution of the genetic algorithm. In their model, user preference is learned based on the inputs made in the first iterations of the interactive genetic algorithm. The machine learning model was only proposed in this article. The model implementation and experimental evaluation were reported in a more recent article (Araújo et al., 2016). Their approach differs significantly to the one proposed in this thesis. In their approach, the underlying optimization problem was solved using the genetic algorithm, i.e., machine learning was solely applied to predict the revenue of the items in the instance based on the users preference. In turn, VS learns how to solve the underlying optimization problem using an exact algorithm as a reference.

Applying VS to the NRP aims to provide a good approximation method that is fully scalable, with fast execution times. The goal is not to provide a state-of-the-art algorithm for the NRP, but to show that it is possible to solve the problem with high accuracy using machine learning.

4.2 VS design for the NRP

The general framework of VS (presented in Section 2.3) needs to be instantiated to the specific problem being solved. In the case of the NRP, a dataset of instances solved by an exact algorithm is used to train VS. Figure 4.1 outlines the process used to build a training set for the NRP. Each problem instance in the dataset is iteratively processed and each requirement is treated as a separate observation during the training phase of VS. Different combinations of features can be considered during training. In fact, different combinations were analyzed and the experimental results are reported in Section 4.4.2.1. The training set generation process depicted in Figure 4.1 is described using the combination of features that achieved the best results in these experiments. The training vector corresponding to one requirement includes the following features: the cost of the requirement, the revenue the requirement renders, and the total budget (which is fixed for all requirements in a given problem instance). The classification label is a binary value, indicating whether the requirement is to be included (1) or not (0) in the next software release, according to the reference algorithm. Each problem instance contributes to the

training set with as many observations as the number of requirements in the instance. This allows drastically reducing the number of reference solutions required in the learning process.

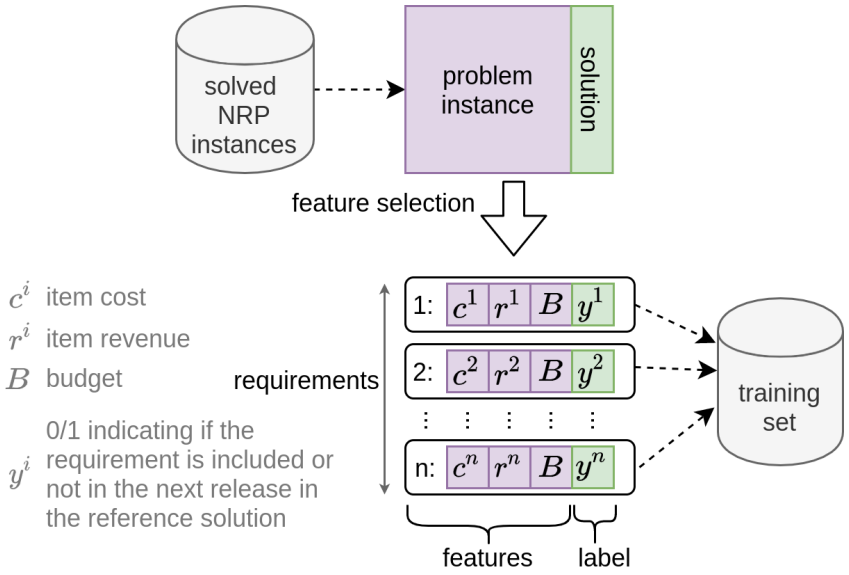


Figure 4.1: VS training set generation for the NRP.

After training the machine learning classifier, VS can handle new, previously unknown, and even bigger instances than those used for learning. Figure 4.2 outlines the workflow of VS when solving the NRP. As described in Section 2.3, the execution of VS is comprised of two phases: *prediction* and *improvement*. In the prediction phase, VS receives as input an unknown NRP instance to solve. Since the machine learning classifier is trained considering each requirement individually, several copies of the same classifier can be spawned, forming a pool, splitting the new problem instance and making predictions for each variable in parallel. Potentially, each requirement in the problem instance can be handled by a different copy of the same classifier. The output of each classifier corresponds to the probability of including the given requirement in the next software release. The results computed by each classifier in the pool are gathered to form a single vector that holds the probability of including each requirement in the next software release.

During the improvement phase, the probability vector built in the prediction phase is used to generate different candidate solutions. Several strategies to build these candidate solutions can be devised. For the NRP, a generic method that performs random samples was used, guided by the probabilities

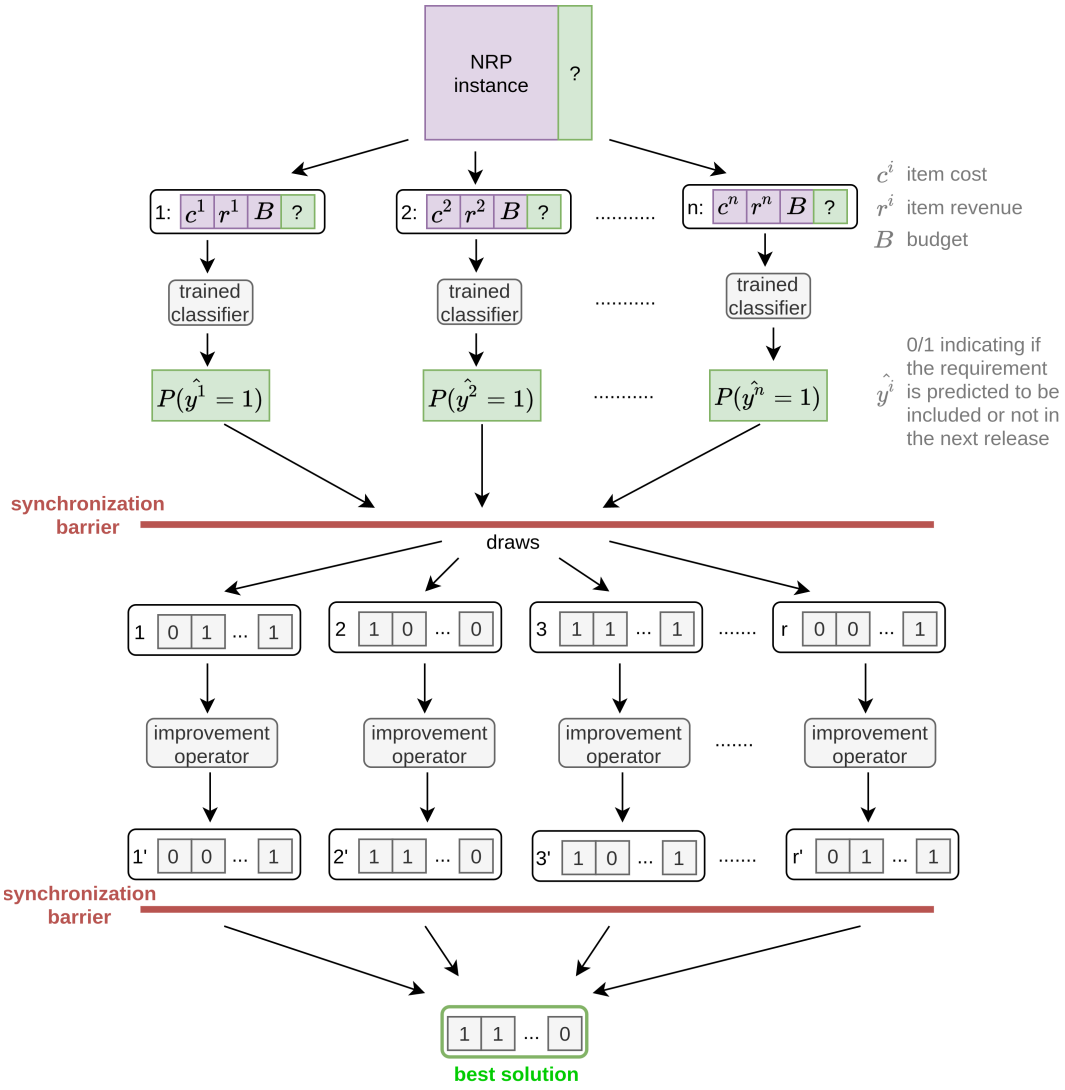


Figure 4.2: VS workflow for the NRP.

of including each requirement as reported by the machine learning classifiers. Each generated solution is then subject to an improvement operator, which aims to modify the solution to achieve better results. This phase also plays an important role in specific problem instances where the prediction phase may perform poorly. The improvement phase is also subject to massive parallelism since one candidate solution can be generated and improved per computing resource available. It is worth noting that the generated solutions may be infeasible, i.e., they may not satisfy the budget constraint. Therefore, during the improvement phase, it is necessary to include a correction operator to ensure the feasibility of the computed solutions. Several alternatives for improvement and correction operators were taken into account, which are described in the following section.

Algorithm 1 outlines a pseudocode of the VS design applied to solve the NRP. The loop at lines 3 to 5 corresponds to the prediction phase, where the probability of including each requirement is predicted based on its cost and revenue and the total budget available. The loop at lines 6 to 9 corresponds to the improvement phase of VS, where candidate solutions are generated based on the probabilities computed in the prediction phase and are further refined using improvement operators. Finally, the best generated solution is returned (line 10). As mentioned earlier, it is straightforward to parallelize both loops. Despite the massively-parallel nature of VS, only a sequential implementation is presented for the NRP since the goal is to evaluate its capability to solve a well-known $\mathcal{NP}$-hard problem. The parallel potential of VS is addressed in Chapter 5 when solving the HCSP.

Algorithm 1: Pseudocode of VS applied to the NRP.

```
   input : instance
1  probability_vector ← []
2  candidate_solutions ← []
3  foreach requirement in instance do
4  |   probability_vector[requirement] ← predict(requirement.cost,
   |     requirement.revenue, instance.budget)
5  end
6  for i ← 0 to candidate_solutions.size − 1 do
7  |   candidate_solutions[i] ← generate_solution(probability_vector)
8  |   improvement_operators(candidate_solutions[i])
9  end
10 return best(candidate_solutions)
```

4.3 VS implementation for the NRP

The implementation details of each phase of VS applied to the NRP are presented next.

4.3.1 Prediction phase

The VS implementation for solving the NRP used SVMs as supervised machine learning classifiers. The training set was built using problem instances solved by the Nemhauser-Ullmann algorithm, which computes exact solutions for the NRP (Nemhauser and Ullmann, 1969; Harman et al., 2014). The LIBSVM framework (Chang and Lin, 2011) was used for the SVM classifier, using the RBF kernel to map vectors into a higher dimensional space.

4.3.2 Improvement phase

Two different proposals were implemented for the improvement phase, which are described next and are later compared and evaluated.

4.3.2.1 LS and corrections

The first proposed scheme for the improvement phase was to apply a simple LS heuristic to each generated solution. This LS operator simply performs random modifications to the candidate solution to exclude or include requirements from the next software release. In each step of the LS, a randomly-chosen bit of the candidate solution is flipped, the new solution is evaluated using a score assignment function, and the LS continues from that solution if an improvement is found. Algorithm 2 describes the score assignment function used to guide the LS. The function considers a solution with total cost C, total revenue R, and an overspent budget $O = C - B$, where B is the total budget. C, R, and O are scaled using the minimum and maximum cost and revenue values in the problem instance. A constant $f > 0$ is used as a penalty factor for solutions that exceed the budget, while $m \in (0, 1)$ is used to define the maximum overspent budget allowed for a solution to be considered by the LS operator.

Algorithm 2: Score assignment for solutions during the LS.

input : solution, instance

1 scale(C, R, O, B, instance)
2 **if** $O \leq 0$ **then return** R
3 **else if** $O \leq m \cdot B$ **then return** $C - f \cdot O$
4 **else return** $-O$

The score assignment function was devised to allow the LS to explore infeasible solutions with a total cost exceeding the budget constraint by up to a factor of m, but penalizing such solutions in order to guide the search towards feasible solutions. Since the solution returned by the LS operator might be infeasible, it is necessary to include a correction operator to guarantee feasibility. Two different correction approaches were considered:

1. *revenue correction*: iteratively removes the requirement with the lowest revenue until the total cost of the solution is not greater than the budget.
2. *cost correction*: iteratively searches for requirements with costs equal to or greater than the amount spent over the budget, removing the requirement with the lowest cost among them. If no requirement satisfies this condition, the requirement with the overall greatest cost is removed.

4.3.2.2 Greedy correction and improvement

Alternative operators for the correction and improvement of solutions, inspired by a popular greedy strategy for knapsack problems, were also implemented and are described next.

- *greedy correction:* while the total cost of the candidate solution exceeds the budget, the requirement with the lowest revenue/cost ratio is iteratively removed to correct infeasible solutions.
- *greedy improvement:* while the total cost of the candidate solution is within the budget (i.e., there is still budget available to add requirements to the next software release), iteratively add the leftover requirements that fit, one by one, in descending order of revenue/cost ratio.

4.4 Experimental analysis

This section reports the experimental analysis of VS when solving the NRP modeled as a 0/1-KP. The NRP instances used in the experiments are described and the results of each phase in VS (i.e., training, prediction, and improvement) are reported and discussed.

4.4.1 Problem instances

VS was trained and evaluated using a standard benchmark of NRP instances. These instances were solved to optimality using the Nemhauser-Ullmann algorithm (Nemhauser and Ullmann, 1969). The benchmark is comprised of problem instances with different sizes (i.e., number of requirements) and Pearson correlations between cost and revenue of the requirements. Pearson correlation is a measure that is usually applied to characterize the difficulty of solving an NRP instance (Harman et al., 2014). The benchmark includes a total of 50 datasets, each with 300 instances with varying size and Pearson correlation.[1].

Within each dataset, instance sizes vary from 100 to 1500 requirements (stepsize: 100). For each instance size, Pearson correlation between cost and revenue of requirements varies from 0.0 to 0.95 (stepsize: 0.05). Figures 4.3 and 4.4 show two examples of the relation between cost and revenue in NRP instances with different Pearson correlation.

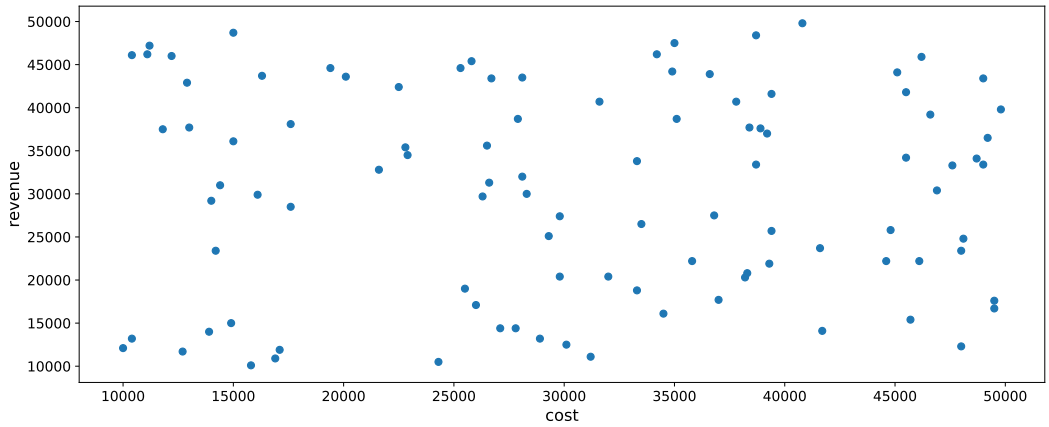


Figure 4.3: Relation between cost and revenue in a sample NRP instance with Pearson correlation = 0.00.

[1]The benchmark is publicly available at ucase.uca.es/nrp

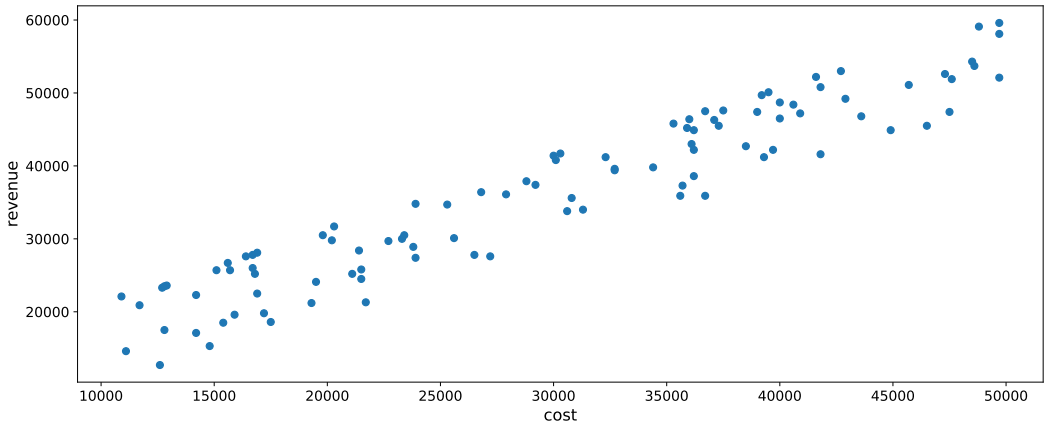


Figure 4.4: Relation between cost and revenue in a sample NRP instance with Pearson correlation = 0.95.

Out of the 50 datasets in the benchmark, dataset #1 was used to train VS, datasets #2 through #5 were used to define the size of the training set and for feature selection, and datasets #6 to #15 were used for the experimental evaluation of VS.

4.4.2 SVM training

Training experiments were performed following an incremental approach, focused on studying different features of the problem, relations between them, and parameter values of the training method. The goal of the study was to determine the configuration of parameters and the combination of features that allowed achieving the best accuracy in the prediction phase of VS for the NRP. Accuracy measures the number of correct predictions out of all predictions made. A three-step analysis was performed, which is described next.

4.4.2.1 First step: study of the input features

In this first step, three different feature configurations were compared.

- *C1*: requirement cost, requirement revenue, and budget (3 features).
- *C2*: requirement cost, requirement revenue, and ratio between the budget and the total number of requirements in the instance (3 features).
- *C3*: requirement cost, requirement revenue, budget, and total number of requirements in the instance (4 features).

The average accuracy (i.e., the percentage of accurate predictions of the trained SVM compared to the optimal solution) for each dataset was evaluated using the three feature configurations. Results showed minor differences among the different feature configurations considered, with accuracy values between 89.4% and 89.7%. Consequently, configuration *C1* was chosen for the remainder of the experimental analysis due to its simplicity.

4.4.2.2 Second step: study of the number of training observations

A study of the number of observations used to train the SVM was performed. Table 4.1 shows the prediction accuracy when varying the number of observations of dataset #1 used during training. Training with 15% of dataset #1 resulted in a 31% improvement in prediction accuracy when compared to using only 10%. However, increasing the size of the training set beyond 15% of dataset #1 resulted in marginal accuracy improvements. Since training times increase drastically with larger training sets, a SVM trained with 15% of dataset #1 was used for the remainder of the experimental evaluation.

Table 4.1: SVM accuracy for different training set sizes.

	number of observations (% of the total)				
	252000 (100%)	126000 (50%)	63000 (25%)	37800 (15%)	25200 (10%)
dataset #2	89.6%	89.5%	89.4%	89.4%	58.0%
dataset #3	89.5%	89.4%	89.3%	89.3%	57.9%
dataset #4	89.6%	89.5%	89.3%	89.3%	58.0%
dataset #5	89.7%	89.6%	89.4%	89.4%	58.0%

4.4.2.3 Third step: parameter configuration

SVM and RBF kernel parameters were configured (parameters C and γ, respectively). For this purpose, cross-validation was performed over a set of 5000 observations randomly selected from the training set. Parameter configuration was performed using the script provided by the LIBSVM framework, which performs a grid-search of parameters. Five-fold cross-validation was performed in a grid defined by $C \in [2^{-5}, 2^{15}]$ (stepsize: 2^2) and $\gamma \in [2^3, 2^{-15}]$ (stepsize: 2^{-2}). Results show that the best accuracy values were computed with $C = 2^{13} = 8192$ and $\gamma = 2^{-1} = 0.5$. Average accuracy values before and

after cross-validation are reported in Table 4.2. An improvement of $\sim 1\%$ in datasets #2 through #5 is achieved after the parameter configuration.

Table 4.2: Average accuracy before and after parameter configuration.

	before	*after*
dataset #2	89.4%	90.4%
dataset #3	89.3%	90.5%
dataset #4	89.3%	90.5%
dataset #5	89.4%	90.5%

4.4.3 Prediction phase

Once the training phase was completed, VS was evaluated using the unseen problem instances from datasets #6 through #15. The experimental evaluation was performed using the best configuration of features, training size, and parameters, which are summarized in Table 4.3.

Table 4.3: Configuration for the experimental evaluation of VS in NRP.

parameter	*value*
feature vector	⟨requirement cost, requirement revenue, budget⟩
training set size	37800
C	8192
γ	0.5

Firstly, the study focused on the prediction phase of VS. Boxplot in Figure 4.5 shows the accuracy achieved by the SVM with problem instances grouped by size. The accuracy is defined as the percentage of variables that are correctly predicted when comparing to the optimal solution provided by the Nemhauser-Ullmann algorithm. Analogously, Figure 4.6 shows the accuracy values achieved when grouping instances by the revenue/cost Pearson correlation of their requirements. The notches in the boxes display the variability of the median between samples. If the notches of two boxes are not overlapped, then it means that there are statistically significant differences in the data with 95% of confidence.

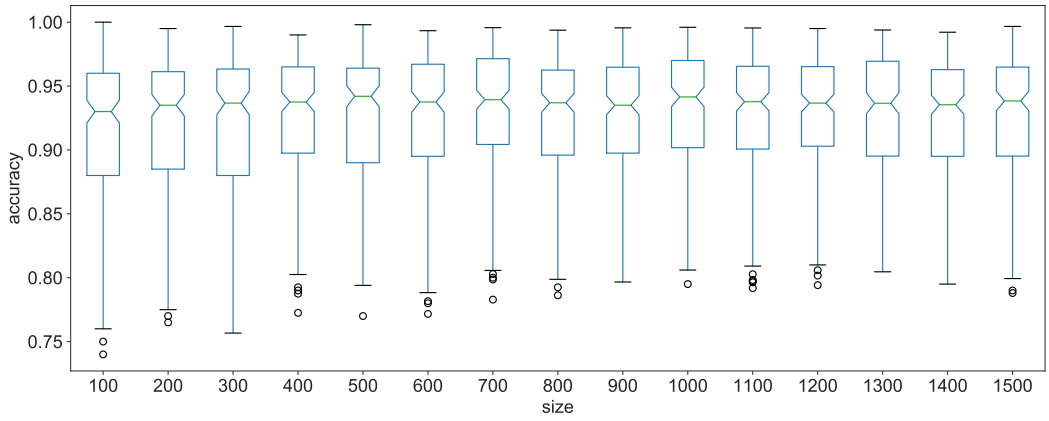


Figure 4.5: SVM accuracy to predict the optimal solution with varying problem size.

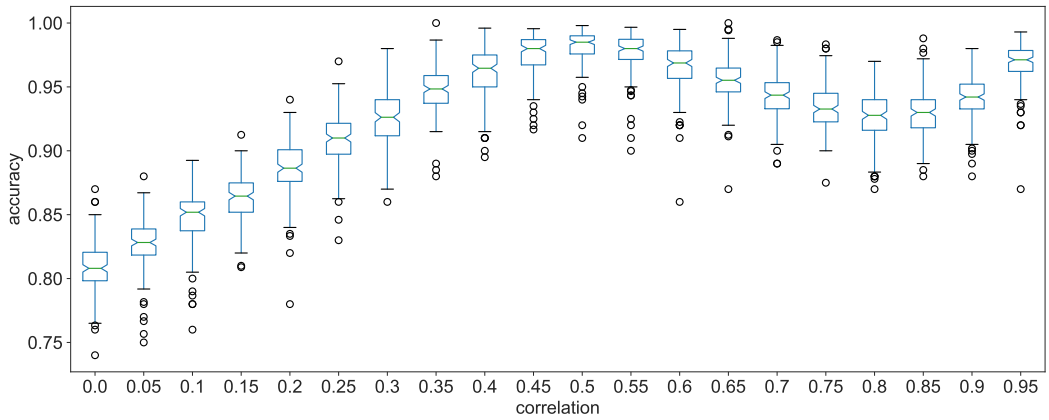


Figure 4.6: SVM accuracy to predict the optimal solution with varying correlation.

The median accuracy achieved by the SVM was over 90% for all problem sizes studied, with no significant differences among instances of different size. This might be explained due to the prediction scheme of VS, where each requirement is considered individually. When grouping instances by revenue/cost correlation, it is noticeable that instances with a correlation of 0.5 were the simplest to predict for the SVM, with a median accuracy of over 97%. In the worst case, when the revenue/cost correlation is 0.0, the median accuracy of the SVM was still above 80%. Overall, the mean accuracy for all studied instances was 92.3%, with a standard deviation of 5.3%. The worst prediction was made for an instance of size 100 and correlation 0.0, with a prediction accuracy of 74.0%. The best prediction accuracy was achieved in two instances of size 100, with correlations 0.35 and 0.65, where the prediction accuracy was 100% (i.e., the optimal solution was predicted).

In order to measure the contribution of the prediction phase of VS, the predicted solutions were evaluated prior to the application of the improvement operators. As explained in Section 4.2, predicted solutions might be infeasible (when the sum of the costs of the requirements exceeds the budget). Thus, in some cases, it was necessary to apply a correction scheme to ensure solution feasibility. For this experiment, the greedy correction described in Section 4.3.2.2 was applied, without using any of the improvement operators. Results show that no corrections at all were needed for over 58% of the studied problem instances. On average, only 2.2% of the requirements in a given instance needed correction. Figures 4.7 and 4.8 show the ratio to the optimum achieved in the prediction phase of VS with varying problem size and correlation, respectively. The ratio to the optima is defined as the quotient between the revenue of the computed solution and the revenue of the optimal solution.

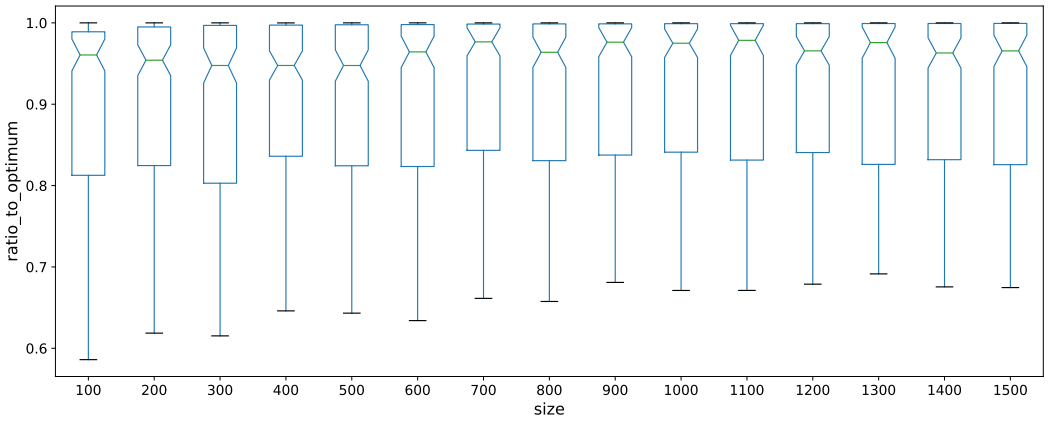


Figure 4.7: Ratio to optimum of SVM predictions with varying problem size.

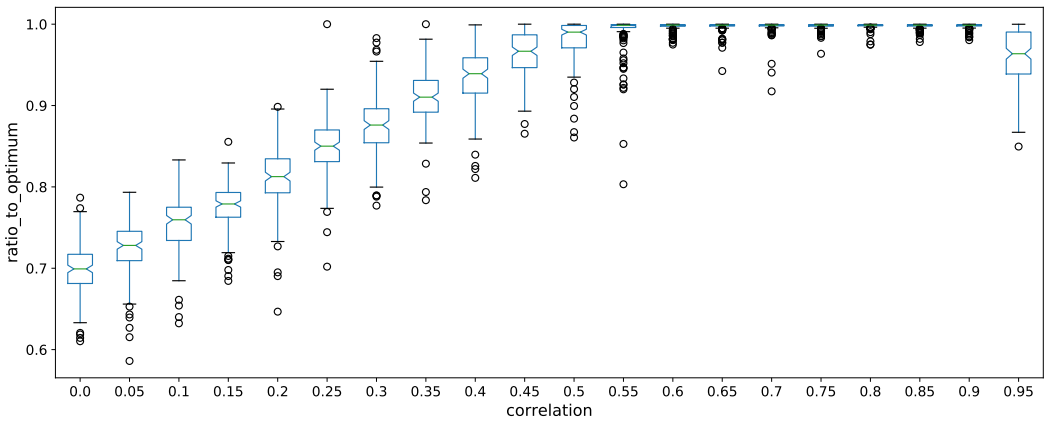


Figure 4.8: Ratio to optimum of SVM predictions with varying correlation.

Results show that the prediction phase of VS computed good quality solutions in most problem instances. On average, solutions computed based only on the prediction phase of VS differed in 9% from the known optima. No significant differences were noticed among the quality of solutions when grouping problem instances by size, with predicted solutions within 10% of the optimal value in median. When grouping problem instances by correlation, results show that the SVM predictions were able to compute better solutions for instances with larger correlation. This result is consistent with the previous analysis of prediction accuracy.

4.4.4 Improvement phase

The experimental results when using the different strategies for the improvement phase are presented next.

4.4.4.1 Corrections and LS

The ratio to the optima was used as a metric to evaluate the results computed by VS. Figures 4.9 and 4.10 show the average ratio to optima achieved when applying only the revenue correction, only the cost correction, the LS followed by the revenue correction, and the LS followed by the cost correction. Average results are presented when grouping instances by size (Figure 4.9) and by revenue/cost correlation (Figure 4.10). Results correspond to 30 independent executions of each problem instance. The LS was executed for 1000 steps. The score function of the LS used the following parameters: $m = 0.2$ and $f = 2$, thus allowing the LS operator to explore solutions that exceeded the budget constraint by up to 20%.

Results show that VS computed accurate results for the studied instances. When grouping instances by size, solutions computed by VS were, on average, only 3% worse than the known optima. When only the correction schemes were applied (without the LS), VS was still able to compute accurate solutions, within 10% from the optima. When grouping instances by correlation, it is particularly interesting to notice that VS was able to solve to optimality instances with correlations of 0.5 and 0.95. On average for all correlations, VS differed by 3.15% from the known optima when using the LS followed by the revenue correction and by 3.20% when using the LS with the cost correction.

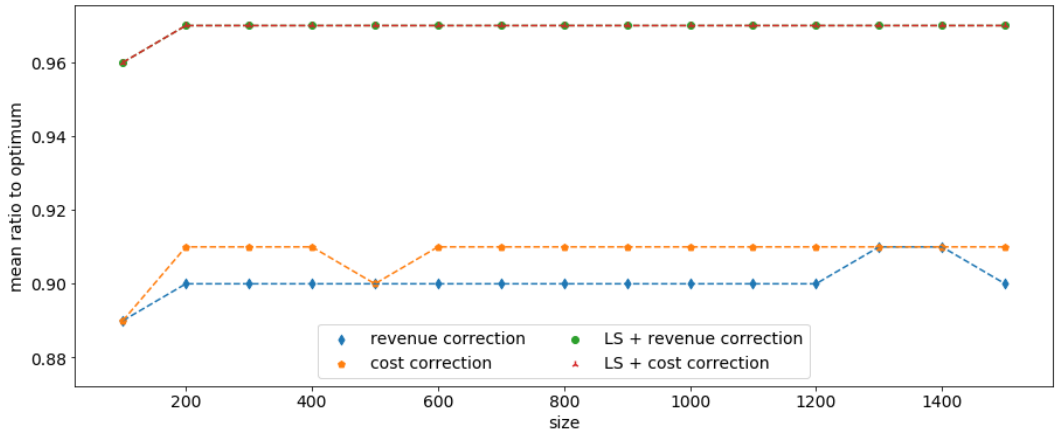


Figure 4.9: Average ratio to optimum grouped by problem size with different improvement operators.

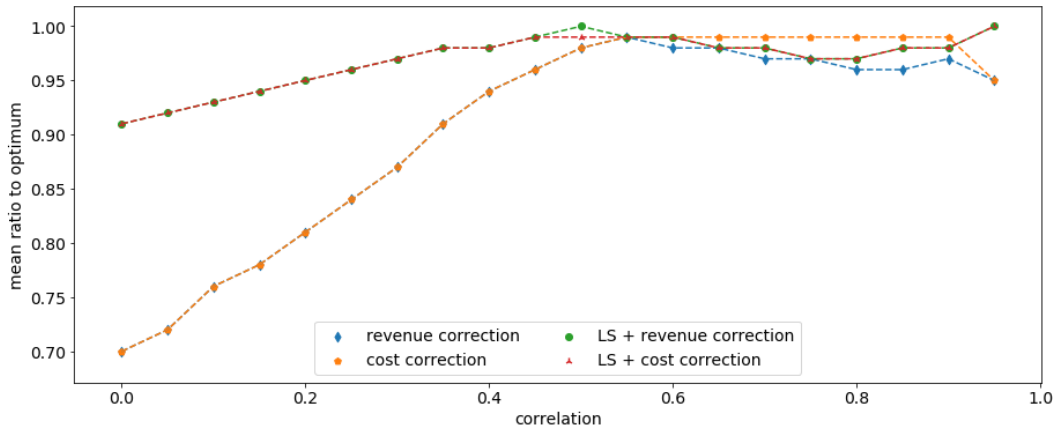


Figure 4.10: Average ratio to optimum grouped by revenue/cost correlation with different improvement operators.

4.4.4.2 Greedy correction and improvement

Finally, the results achieved by VS when using the greedy correction and improvement are presented. Figure 4.11 presents the achieved ratio to the optimum when grouping instances by size. Similarly, Figure 4.12 shows the computed results when grouping instances by the revenue/cost correlation of their requirements.

The results achieved when using the greedy and correction improvements were within 1% from the optima in all the instances under study. On average, computed results were within 0.04% from the optima. In the worst case, for an instance of size 100 and correlation 0.7, the solution computed by VS differed in only 0.66% from the known optima. Additionally, the optimal solution was

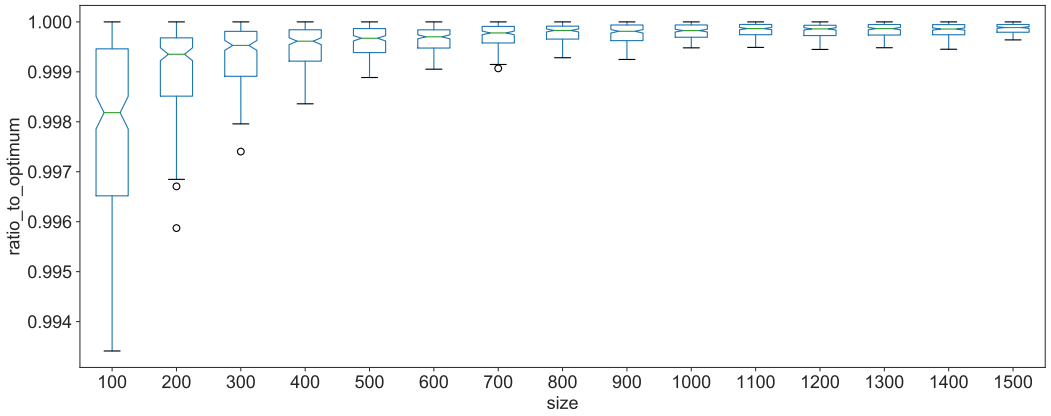


Figure 4.11: Average ratio to optimum with varying problem size using greedy correction and improvement.

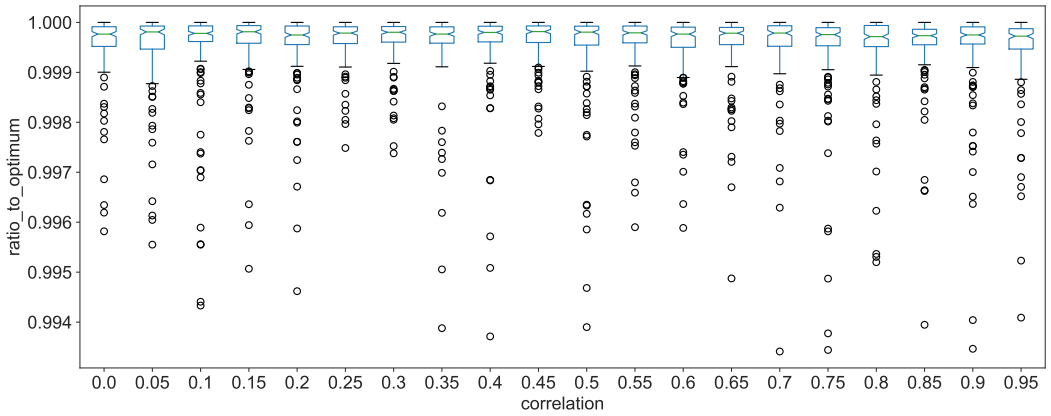


Figure 4.12: Average ratio to optimum with varying revenue/cost correlation using greedy correction and improvement.

computed for 5.5% of all problem instances studied. VS solved to optimality instances of all sizes and correlations. The group of instances having size 100 and correlation 0.40 was the one with the highest number of instances that were optimally solved. When looking at instances by size, VS performed better on larger instances. The median ratio to the optima was less than 0.2% for all problem sizes studied. No significant differences were noticed among problem instances when grouping by revenue/cost correlation. It is worth noting that this correction and improvement schemes are based on a well-known greedy heuristic for the 0/1-KP, thus providing valuable domain-specific information to VS, unlike the LS and corrections presented earlier.

The greedy correction and improvement operators, which achieved the best overall results, were applied to random solutions to further show the contribu-

tion of the prediction phase of VS. The goal of this experiment was to show that the prediction phase plays an important role in computing a good quality initial solution which is then improved using the improvement operators. Figures 4.13 and 4.14 show the ratio to the optimum achieved when applying the greedy correction and improvement to randomly-generated solutions on problem instances with varying size and correlation, respectively.

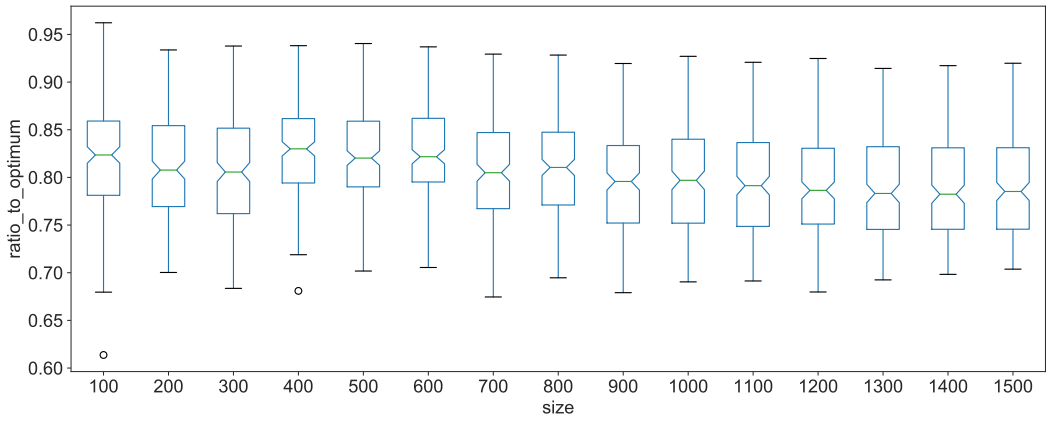


Figure 4.13: Average ratio to optimum with varying problem size starting from random solution and using greedy correction and improvement.

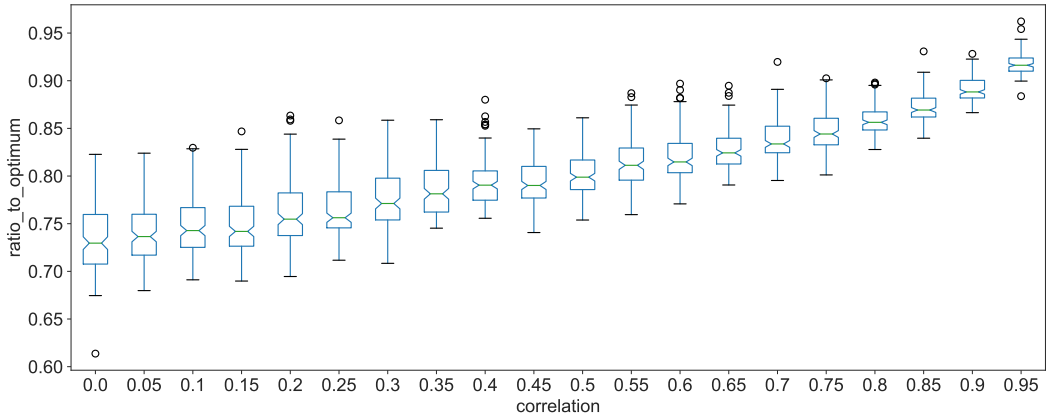


Figure 4.14: Average ratio to optimum with varying revenue/cost correlation starting from random solution and using greedy correction and improvement.

Results achieved when starting from a randomly-generated solution are, on average, 20% away from the optima. The importance of both phases of VS can be noticed when comparing these poor results against those reported in Figures 4.11 and 4.12.

4.5 Conclusions

This chapter presented how VS can be used to efficiently solve a complex problem in RE: the basic independent NRP, which can be formulated as a 0/1-KP. A thorough study of the training and prediction phases of VS was presented, and five different variants for the improvement phase of VS were analyzed for the problem. Experimental evaluation was performed using a publicly-available benchmark of problem instances of varying size and correlation between the revenue and the cost of the requirements, which is a measure of instance difficulty. The Nemhauser-Ullmann algorithm, an exact method for the problem, was used as the reference algorithm for VS.

Firstly, a brief comparison of different feature configurations was performed, which showed no significant differences among the considered options. Secondly, a study on the training set size required for accurate learning was presented. Smaller subsets of 10%, 15%, 25%, 50%, and 100% of the observations in dataset #1 in the benchmark were considered. Results showed that a training set built using 15% of the dataset was enough to make accurate predictions. Beyond that percentage, only marginal improvements were observed. Thirdly, model parameters were configured using cross-validation. When considering only the efficacy of the prediction phase on unseen instances, VS was able to predict the exact solution with a median accuracy larger than 90% when grouping instances by their size and larger than 80% when grouping instances by their revenue/cost correlation.

The improvement phase in VS helps further refining the solutions generated in the prediction step. Experimental results showed that VS can compute highly-accurate solutions. Among the five improvement strategies presented in this chapter, the simplest variant (a greedy mechanism that first corrects the solution and then improves it, based on the revenue/cost ratio of requirements) was the one that achieved the best results. The solutions computed by the VS version implementing this greedy mechanism were within 1% from the optima in all studied instances. Furthermore, VS was able to generate the optimal solution in many cases, and the computed solutions were within 0.2% (in median) of the known optima computed by the exact algorithm used as a reference. It was also observed that, interestingly, difficult instances for the reference algorithm were not necessarily difficult to solve for VS.

Chapter 5

Virtual Savant for the Heterogeneous Computing Scheduling Problem

This chapter presents the application of VS to the HCSP. Section 5.1 introduces the HCSP, a widely studied scheduling problem in the context of HPC. Then, Section 5.2 presents the VS implementation to solve the HCSP. The experimental analysis is outlined in Section 5.3 and, finally, conclusions are presented in Section 5.4.

5.1 Heterogeneous Computing Scheduling Problem

This section presents the HCSP, its mathematical formulation, and a brief review of the related literature on the problem.

5.1.1 HCSP overview

A heterogeneous computing system is a coordinated set of processing elements, often called resources, processors or simply machines, interconnected by a network, which can work cooperatively to solve complex problems. The heterogeneous aspect refers to the variable computational capabilities of the resources, most commonly the CPU processing power, but often other features such as RAM or external storage (Khokhar et al., 1993). Taking into account the

diverse computing capabilities of heterogeneous computing systems, finding suitable task-to-machine assignments is a key issue to achieve an appropriate load-balancing. Generally, the goal of the scheduling problem is to find an assignment that optimizes a given metric related to efficiency, economic profit, or quality of service offered to the users of the system.

In the context of HPC, finding accurate schedules has an important effect on the performance of systems for both users—since it improves the quality of service—and providers—contributing to efficiently use the available resources. Traditional scheduling problems are $\mathcal{NP}$-hard (Garey and Johnson, 1990), thus, classic exact methods are only useful for solving problem instances of reduced size. When dealing with large computing environments, approximate algorithms emerge as promising methods for solving scheduling problems. These methods allow computing accurate solutions in reasonable execution times, which usually satisfy the efficiency requirements of real-life scenarios.

Computing the schedule for all arriving tasks directly impacts the response time offered to users. The response time of the system is defined as the time since the user submits his/her job until its execution is started. For this reason, modern schedulers are simple heuristics that can offer acceptable solutions in short computation times (generally less than a second). In contrast, more complex optimization methods, such as metaheuristics, offer more accurate solutions to the problem, but at the cost of considerably longer response times. Therefore, there is a need for new methodologies that can quickly find high-quality schedules.

5.1.2 HCSP formulation

The HCSP considers a heterogeneous computing system comprised of several resources (i.e., *machines*) and a set of *tasks* with variable computational requirements to be executed in the system. A task is defined as an atomic workload unit, i.e., it must be executed without interruptions and cannot be split into smaller chunks, which corresponds to a non-preemptive scheduling model. The execution time of any individual task varies from one machine to another and is assumed to be known beforehand, following a static scheduling approach. The HCSP proposes finding a task-to-machine assignment that optimizes some quality metric.

Several variants of the HCSP have been proposed by considering task-to-machine assignments that optimize different quality metrics (Nesmachnow et al., 2010; Nesmachnow, 2013; Iturriaga et al., 2014). The scheduling problem addressed in this thesis focuses on optimizing the makespan, a well-known optimization criterion related to the productivity of a computing system. Makespan is defined as the time between the start of the first task (in the set of tasks to be executed) and the completion of the last task. Makespan is considered as a measure of productivity (i.e., throughput) of computing systems. This problem model is suitable for applications following the bag-of-tasks approach for independent executions as well as for the scheduling of tasks submitted by different users, a usual scenario on modern cluster, grid, and cloud computing platforms.

The mathematical formulation for the HCSP considers:

- A set of tasks $T = \{t_1, \ldots, t_n\}$ to be scheduled and executed on the system.
- A set of heterogeneous machines $M = \{m_1, \ldots, m_m\}$.
- A function $ET : T \times M \to \mathcal{R}^+$ where $ET(t_i, m_j)$ indicates the execution time of task t_i on machine m_j.

The HCSP proposes finding an assignment function $f : T \to M$ that minimizes the makespan, defined by Equation 5.1.

$$\text{makespan} = \max_{m_j \in M} \sum_{t_i \in T, f(t_i) = m_j} ET(t_i, m_j) \tag{5.1}$$

The proposed model does not account for dependencies among tasks: the problem formulation assumes that all tasks can be independently executed, without considering the execution order. Even though more general formulations of the HCSP exist—e.g., accounting for task dependencies and other objectives (Dorronsoro et al., 2014)—the independent task model is highly relevant, especially in distributed computing environments. Independent-task applications frequently appear in many lines of scientific research as well as in shared infrastructures, where different users submit tasks to be executed. Thus, the relevance of the HCSP version dealt with in this thesis is justified due to its significance in realistic distributed computing environments.

5.1.3 Related work

In the last twenty years, research on the HCSP has been increasing due to the popularity of modern parallel and distributed computing systems. Heuristic and metaheuristics approaches are good candidates to address the HCSP, especially in real-life scenarios where schedulers are expected to provide accurate solutions in short execution times. A large number of heuristics (Braun et al., 2001; Ibarra and Kim, 1977) and metaheuristics (Nesmachnow et al., 2010, 2012; Wang et al., 1997; Pinel and Dorronsoro, 2014; Pinel et al., 2010; Xhafa et al., 2012) have been proposed for efficiently solving the HCSP. Among other metaheuristic techniques, EAs have been widely applied for solving scheduling problems in heterogeneous computing systems.

Braun et al. (2001) presented a systematic comparison of eleven scheduling heuristics for the HCSP, including an EA and a hybrid algorithm that combined an EA with Simulated Annealing. The proposed algorithms were seeded during the population initialization to significantly improve the search. Both methods were able to obtain the best-known makespan values at that time for the studied HCSP scenarios.

Duran and Xhafa (2006) studied a steady-state GA and the Struggle GA for the HCSP. Both GAs outperformed previous results in more than half of the studied instances. A memetic algorithm (MA) proposed by Xhafa (2007) included subordinate LS methods to find high-quality solutions in short execution times. Later, the structured population of a cellular MA was used to control the trade-off between the exploitation and exploration of the HCSP solution space (Xhafa et al., 2008). Using a seeded population initialization and three LS methods, the cellular MA outperformed previous GA results for half of the instances from Braun et al. (2001).

Recent works explored applying the use of machine learning methods for estimating the performance of applications on different heterogeneous resources (Nemirovsky et al., 2017), for performance estimation and resource selection (Zhao et al., 2009), and for predicting the speedup of CPU/GPU kernels (Wen et al., 2014). However, according to the literature reviewed, no other approaches for automatically designing schedulers using machine learning techniques exist besides VS (Dorronsoro and Pinel, 2017; Pinel et al., 2018, 2013; Pinel and Dorronsoro, 2014). The VS implementation for HCSP uses MinMin as a reference algorithm, which is presented next.

MinMin is one of the most widely used methods for solving the HCSP (Luo et al., 2007). MinMin is a two-phase greedy scheduler that greedily picks the task that can be completed the soonest. Starting from a set U of all unmapped tasks, MinMin determines the machine that provides the Minimum Completion Time (MCT) for each task in U, and assigns the task with the minimum overall MCT to its best machine. The mapped task is removed from U, and the process is repeated until all tasks are mapped. MinMin does not consider a single task at a time but all the unmapped tasks sorted by MCT. The availability status of the machines is updated accordingly after each assignment. The idea behind Minmin is to select at every time the task from the unscheduled ones that causes the minimum increase in the overall makespan value. Therefore, the algorithm focuses on balancing the load among all the processors. This procedure generally leads to more balanced schedules and better makespan values than other heuristics, since tasks that can be completed the earliest are assigned first to the corresponding machine (Braun et al., 2001).

5.2 VS for the HCSP

This section presents the application of VS to the HCSP. Section 5.2.1 outlines the design of VS when solving this problem and Section 5.2.2 presents the details of the parallel implementation.

5.2.1 VS design for the HCSP

The VS implementation for the HCSP uses SVMs as machine learning classifiers. SVMs are trained using MinMin as the reference algorithm. The custom xphi-LIBSVM framework (described in Section 2.4.3.3) was used with RBF as the kernel function. Figure 5.1 outlines the training scheme of VS to solve the HCSP. Each task in the instance is considered individually during the training phase of VS. Therefore, each feature vector holds the execution time of one task on each machine and the classification label corresponds to the machine assigned to that task by the MinMin heuristic. This way, one single MinMin solution yields as many observations as the number of tasks in the problem instance.

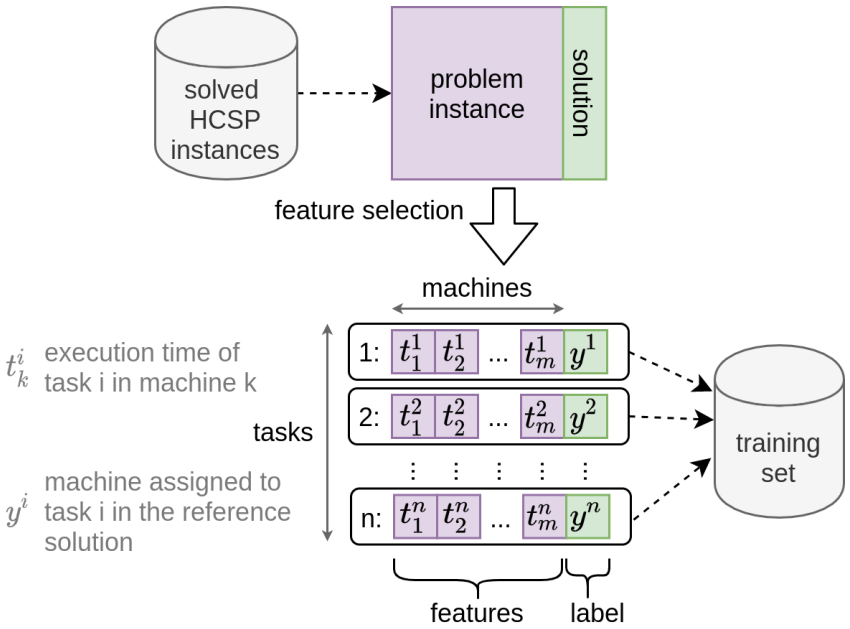


Figure 5.1: VS training set generation for the HCSP.

The complete model of VS to solve the HCSP is presented in Figure 5.2. VS receives as input an HCSP instance, which consists of a matrix with the execution time of each task on each machine. Given that the training phase is done considering each task separately, predictions (i.e., task-to-machine assignments) can be made independently for each task. Thus, multiple copies of the same SVM can be spawned, forming a pool. The problem instance is split among this pool and predictions are made for each task independently, providing VS with a high degree of parallelism. At the finest grain, VS can predict the machine assigned to each task in the problem instance using a different copy of the same SVM classifier. The output of each SVM corresponds to the probability of assigning the task given as input to each of the possible machines. The predictions of all SVMs in the pool are gathered to form a matrix that holds, for each task, its assignment probability on each machine.

Because tasks are independently assigned, VS can scale to problem instances with any number of tasks, without requiring any additional training process. However, since the length of the training vectors depends on the number of machines, different SVMs must be trained to solve problem instances with a varying number of machines. This does not impose a huge limitation on the proposed scheme since, in a real environment, the number of machines has usually a low variability when compared to the number of tasks to schedule.

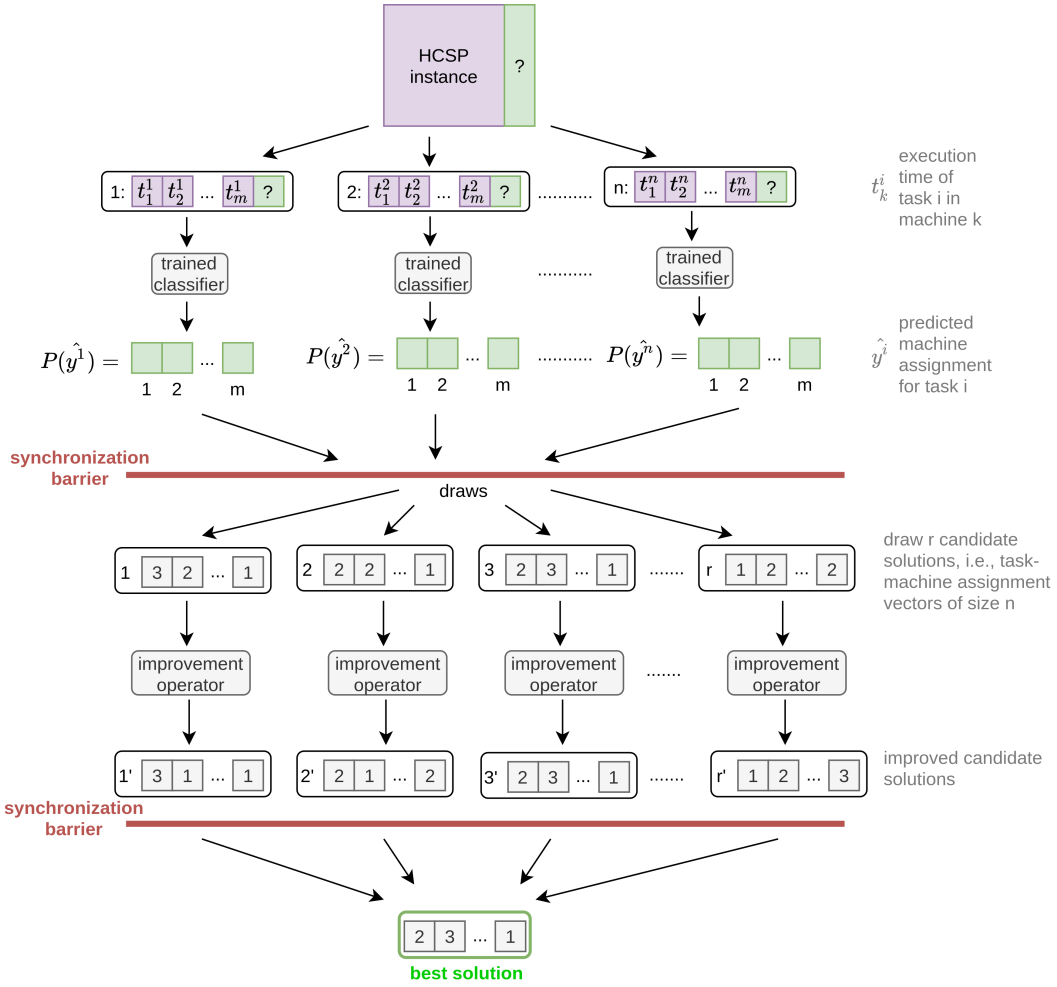


Figure 5.2: VS workflow for the HCSP.

During the improvement phase, the matrix with assignment probabilities—computed in the prediction phase—is used to generate a set of candidate solutions to the problem instance. This set is randomly generated according to the probability of assigning each task to each machine. Then, a simple LS heuristic is applied over each generated solution, which is described in Algorithm 3. The LS heuristic iteratively moves a randomly-chosen task from the most loaded machine, i.e., the one with the highest completion time (lines 2–3), to a machine selected among a subset of the least loaded ones (lines 5–12). Parameter $N \in (0, 1]$ controls the size of this subset. The selected machine is the one with the smallest completion time after the task is moved (lines 7–10).

Algorithm 3: LS for the improvement phase of VS in HCSP.

```
1  foreach step in steps do
2      sort machines on ascending completion time
3      task ← random task from last machines
4      best_score ← completion_time(last machines) // makespan
5      foreach mac in ⌈N · #machines⌉ first machines do
6          new_score ← completion_time(mac) + ET(task,mac)
7          if new_score<best_score then
8              best_mac←mac
9              best_score←new_score
10         end
11     end
12     move task to best_mac if any
13 end
```

The improvement phase is also inherently parallel since the improvement of a given generated solution is independent of the others. Thus, one candidate solution can be generated and improved per computing resource available. After all LSs are completed, the overall best solution found is returned.

5.2.2 Parallel implementation of VS for the HCSP

The parallel implementation of VS for HCSP works for both distributed- and shared-memory architectures. It is based on the MPICH implementation of the MPI standard and the OpenMP library. The fact that the two phases of VS that can be executed in parallel (i.e., the prediction and the improvement phases) have no data dependencies allows developing efficient implementations of the model that take advantage of multiple computing resources.

The parallel implementation of VS for HCSP follows the same structure as the design presented in Figure 5.2. A master node launches the prediction phase, which is performed in parallel using as many processes as the number of available computing nodes. Assuming a homogeneous architecture, the number of problem variables assigned to each process (i.e., the number of predictions every process should compute) in order to balance the load of all computing resources is ⌈number of variables/number of cores⌉.

The master node sends to every slave process the information required by the SVM. This information is taken from the problem instance and, in the particular case of the HCSP, corresponds to the execution time on all machines

for each task assigned to each slave. Therefore, the amount of data transmitted to every computing node is equal to the number of variables to be predicted using the SVM times the number of machines in the problem instance times the space needed to store an integer number. If several variables are to be predicted in a given computing node, these predictions are done in parallel, using as many SVM replicas as the number of threads each core can execute.

The master node waits until all processes are finished to build the probabilities matrix. Every process sends, for each assigned variable, the probability of assigning the corresponding task to each machine, so the amount of information is the same as that received from the master at the beginning of the execution. After this matrix of probabilities is built—by just merging all collected results into one single matrix—the master node spawns as many LSs as processes the system can execute in parallel. Each slave process receives the matrix of probabilities from the master, builds a randomly-generated solution according to these probabilities, and iteratively applies the LS algorithm to this candidate solution. Once the slave finish executing the LS algorithm, it returns the best solution found to the master. Among all received solutions, the master node reports the best one as the final result of VS. It is worth noting that, VS increases the number of LSs spawned—therefore the number of computations performed—when the number of computing resources is increased.

5.3 Experimental analysis

Firstly, an overview of the experimental analysis is presented in Section 5.3.1. Then, Section 5.3.2 presents the results of VS when solving a large set of HCSP instances on a many-core computing platform. Afterward, Section 5.3.3 outlines the experimental analysis performed in four different computing platforms when scaling the number of computational resources. Finally, Section 5.3.4 presents the results of VS when solving very large problem instances.

5.3.1 Overview

The experimental evaluation of VS for HCSP focused on studying the accuracy of the computed solutions and the scalability when increasing both the number of computational resources and the size of the problem. Experiments

used SVMs from previous works (Pinel et al., 2013; Pinel and Dorronsoro, 2014), which were trained using the MinMin algorithm as reference. Thus, the experimental analysis does not focus on the training phase of VS but on its execution when solving new, unseen, HCSP instances on different computing infrastructures.

5.3.2 Execution in a many-core environment

This section reports the experimental analysis of VS when solving a set of 180 HCSP instances in a many-core computing infrastructure.

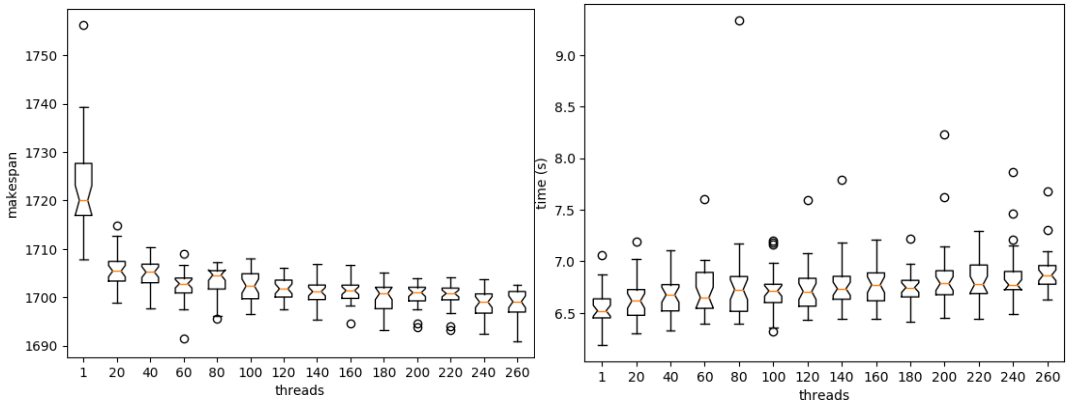
5.3.2.1 Methodology and problem instances

Experiments were performed on a server with an Intel®Xeon Phi™ 7250 processor (68 cores and 64GB RAM). The server was used exclusively during the experiments to accurately measure execution times. As explained in Section 5.2.1, the LS algorithm randomly gets a task from the machine with the highest completion time and assigns it to a machine among the N least loaded ones. The value of N was set to 50% of the total number of machines based on the results from previous works (Pinel et al., 2013; Pinel and Dorronsoro, 2014). The LS operator was set to iterate for 10 000 steps in these experiments since preliminary results showed that no significant improvements were achieved with larger executions.

A set of HCSP instances were generated according to the methodology described by Braun et al. (2001). Tasks were considered as highly heterogeneous and machines were considered consistent (i.e., a machine cannot be faster than another for a given task and slower for a different task). High (*hi*) and low (*lo*) machine heterogeneity configurations were considered in the experimental evaluation. Three problem dimensions were studied (tasks×machines): 128×4, 512×16, and 1024×16, and 30 different problem instances were generated for each combination of problem dimension and machine heterogeneity, totaling 180 problem instances. The notation used to describe each problem instance is: T×M_H_i, where T is the number of tasks, M is the number of machines, H is either *hi* or *lo* indicating high or low machine heterogeneity, and i indicates the instance number ($i \in [1 \dots 30]$).

5.3.2.2 Scalability using parallel processing threads

Boxplots in Figure 5.3 show the distribution of makespan and execution times (in seconds) when varying the number of threads used for the prediction and improvement phases. Results correspond to the 30 independent executions performed with each number of threads on instance *512×16_hi_1* and are representative of the trends observed for other problem instances. Results are summarized in Table 5.1, which reports the best, mean, median, and standard deviation values of makespan and execution times.



(a) makespan **(b)** execution time

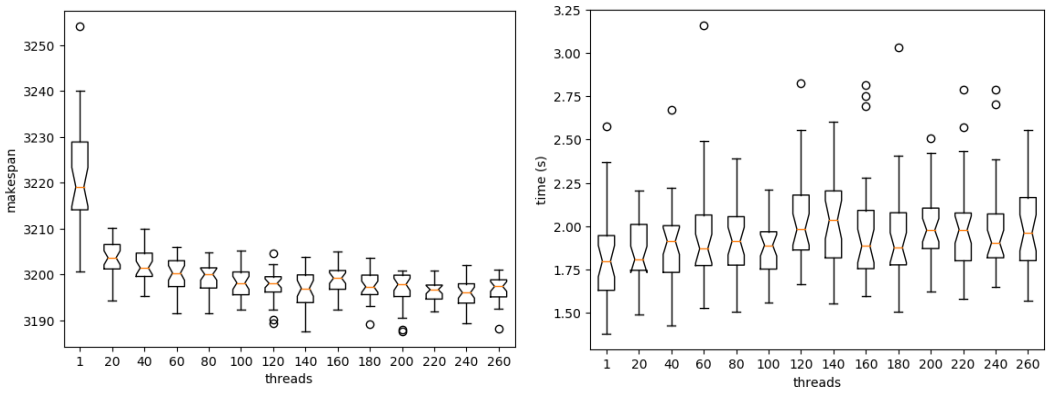
Figure 5.3: Results on HCSP instance *512×16_hi_1* with varying number of threads.

Results show that increasing the number of threads allowed computing better results in terms of makespan since more LSs are performed (1.4% average improvement over the sequential version when using 260 threads). Regarding execution times, the average increase was only 6% when varying from 1 to 260 threads. These results confirm the good scalability properties of VS when increasing the number of computing elements.

Figure 5.4 shows how the results and conclusions obtained for instance *512×16_hi_1* hold in the case of a smaller instance with low heterogeneity, namely *128×4_lo_1*. When comparing Figures 5.3 and 5.4 it can be seen that the trend on the evolution of the makespan of solutions when increasing the number of threads is similar in both cases. In the same way, a similar trend is observed when considering execution times. For the rest of the experiments presented in this section, the number of threads was set to 260, because it allows computing the best results without a negative impact on the execution time.

Table 5.1: Results on HCSP instance *512×16_hi_1* with varying number of threads.

threads	*makespan*				*execution time (s)*			
	best	mean	median	std	best	mean	median	std
1	1707.78	1722.62	1720.03	10.44	6.19	6.55	6.52	0.17
20	1698.94	1705.72	1705.40	3.61	6.30	6.62	6.62	0.21
40	1697.67	1704.99	1705.20	3.05	6.33	6.65	6.67	0.18
60	1691.55	1702.30	1702.80	3.09	6.39	6.71	6.65	0.25
80	1695.61	1703.38	1704.59	2.94	6.39	6.79	6.72	0.52
100	1696.52	1702.02	1702.32	3.06	6.32	6.71	6.71	0.21
120	1697.44	1701.73	1701.74	2.14	6.43	6.73	6.71	0.23
140	1695.41	1700.78	1701.16	2.71	6.44	6.78	6.73	0.25
160	1694.68	1701.38	1701.37	2.31	6.44	6.78	6.77	0.20
180	1693.32	1699.99	1700.78	3.02	6.41	6.73	6.74	0.16
200	1693.81	1700.43	1700.96	2.40	6.45	6.86	6.78	0.34
220	1693.30	1700.25	1700.78	2.44	6.44	6.82	6.77	0.19
240	1692.38	1698.78	1699.01	2.65	6.49	6.85	6.77	0.27
260	1690.91	1698.65	1699.07	2.77	6.62	6.90	6.86	0.21



(a) makespan **(b)** execution time

Figure 5.4: Results on HCSP instance *128×4_lo_1* with varying number of threads.

5.3.2.3 Comparison against MinMin

This section presents a comparison of VS against MinMin, the greedy heuristic used as a reference algorithm. The main goal was to analyze how accurately VS can learn from MinMin to solve the HCSP. The largest instances used during the training phase of VS were of size 512×16. As a preliminary study on the scalability of VS in the problem dimension, VS was executed on larger instances (1024×16) than those used for the training phase. An extended study on the scalability of VS in the problem dimension is presented in Section 5.3.4.

Table 5.2 reports the ratio between the makespan achieved by VS and the makespan of the MinMin solution. Reported results include the best, mean, median, and standard deviation of the ratios achieved on each instance type (30 independent executions of 30 problem instances for each type). Since the goal is to minimize the makespan, ratios lower than 1.0 indicate an improvement of VS over MinMin. Additionally, the last column in the table reports the number of times when VS outperformed MinMin, for every instance type. The distribution of results are also outlined in the boxplot in Figure 5.5.

Table 5.2: Makespan comparison of VS against MinMin for the HCSP.

instance type	$makespan(VS)/makespan(MinMin)$				#improvement
	best	mean	median	std	
128×4_hi	0.85	0.94	0.95	0.02	900/900
128×4_lo	0.93	0.96	0.96	0.01	900/900
512×16_hi	0.89	0.93	0.93	0.02	900/900
512×16_lo	0.92	0.94	0.94	0.01	900/900
1024×16_hi	0.94	0.98	0.98	0.01	853/900
1024×16_lo	0.95	0.97	0.97	0.01	900/900

VS outperformed MinMin on every execution of each problem instance with the same dimensions used for training. The best overall improvement considering all executions was achieved on instances of type *128×4_hi* with a 15% improvement over MinMin. On average, the best improvement was achieved on instances *512×16_hi* with a 7% improvement over MinMin. Additionally, results show that VS was able to accurately solve problem instances of larger dimensions than those used during the training phase. For instances of size 1024×16 (i.e., double the size of the instances used for training) VS was still able to outperform MinMin in most cases, with improvements up to 6%.

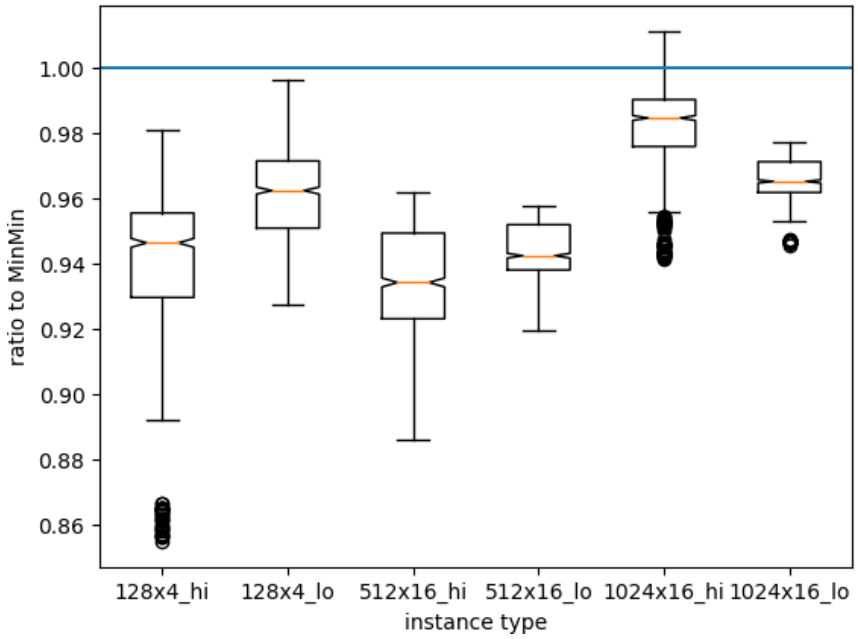


Figure 5.5: Ratio of makespan: VS over MinMin for the HCSP. Values below 1.0 represent the cases when VS outperforms MinMin.

5.3.3 Scalability on different computing platforms

This section reports the experimental results of evaluating the performance of VS when scaling the number of resources used in four different computing platforms.

5.3.3.1 Methodology and problem instances

For this part of the experimental evaluation, two different problem instances of the HCSP were considered: i) instance *512×16*, consisting of 512 tasks and 16 machines, ii) instance *4096×16*, consisting of 4096 tasks and 16 machines. Both problem instances were generated according to the methodology described by Braun et al. (2001), with highly-heterogeneous tasks and machines and consistent machines. The largest instances used during the training phase of VS were of size 512×16. Thus, instance *4096×16* allows further studying the scalability of VS in the problem dimension, solving an instance four times larger than those used for the training phase. A study involving even larger problem instances is presented in Section 5.3.4. For this analysis, the LS was set to perform 1000 steps since preliminary studies showed that this value provided a good compromise between the quality of solutions and the execution times. Parameter N was set to consider 50% of the total number of

machines when moving tasks as defined by previous works (Pinel et al., 2013; Pinel and Dorronsoro, 2014). Results on each architecture are compared in terms of speedup, which is defined as the ratio between the execution time of the sequential implementation and the execution time of its equivalent parallel implementation.

5.3.3.2 Computational platforms

Four different hardware architectures were considered during the experimental evaluation. These architectures include single-, multi-, and many-core computing infrastructures with both shared- and distributed-memory systems. This provided a diverse array of platforms for the experimental evaluation comprised of:

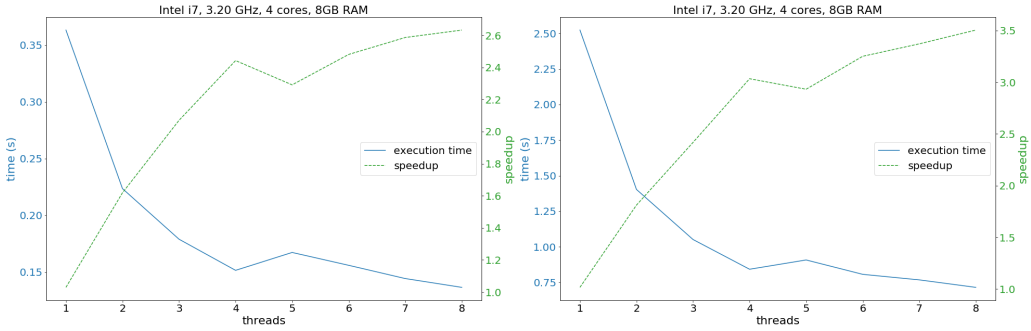
- *Desktop*, which corresponds to a regular desktop PC with an Intel i7 4792MQ quad-core processor with 3.20GHz of peak frequency and 8 GB of RAM.
- *Xeon*, a standalone server comprised of an Intel Xeon E5-2620 CPU with six cores and a maximum peak frequency of 2.6 GHz.
- *Cluster*, which consists of a supercomputing sever composed of three rack-server type-C (c7000) from HP. Each of them contains sixteen nodes bl460c, with two Intel Xeon E5 2670 with 2.60 GHz processors per node. Each of these processors has eight cores, thus it has $48 \times 2 \times 8 = 768$ cores. Nodes are equipped with 128 GB of RAM and are connected through a 10 Gigabit Ethernet.
- *Phi*, comprised of an Intel®Xeon Phi™ 7250 processor, with 1.60 GHz maximum frequency speed and 68 cores, and 48 GB of RAM.

5.3.3.3 Numerical results

The performance of VS was studied on each considered architecture for both HCSP instances studied.

5.3.3.3.1 Desktop architecture. Figure 5.6 shows the execution times of VS on the Desktop architecture when varying the number of threads. For each of the studied instances, the average execution time of 10 independent runs is reported. Additionally, the obtained speedup when using different number

of threads is shown. Results show that the execution time was reduced when increasing the number of threads, due to the parallel capabilities of the VS model. For a correct interpretation of the provided results, it is worth noting that the computational load of the improvement phase does not remain constant since the number of LSs increases with the number of threads. The ideal case for the improvement phase is that the execution time remains almost constant when increasing the number of threads. This means that there is no overhead or bottlenecks when increasing the number of threads, a desirable condition for parallel algorithms. Consequently, the only improvement in execution times due to the use of multiple computing resources can be achieved during the prediction phase of VS.



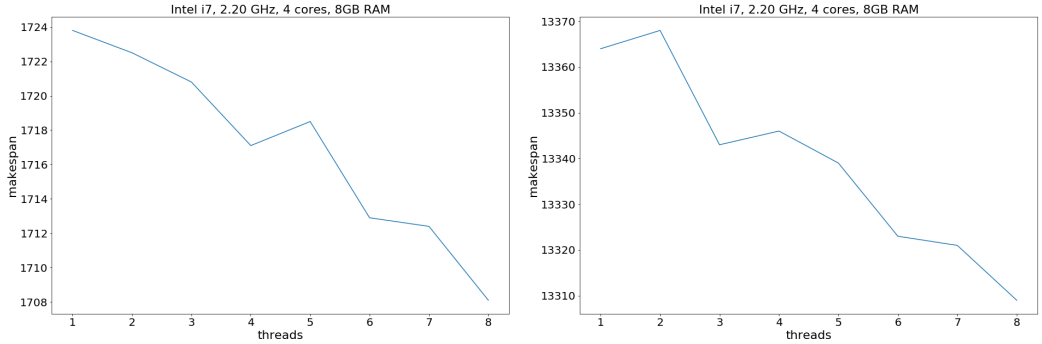
(a) 512×16 **(b)** 4096×16

Figure 5.6: Average execution times with varying number of threads on Desktop architecture.

The maximum speedup achieved in Desktop architecture was 2.63 for instance 512×16 and 3.50 for instance 4096×16. These are highly successful results, taking into account that the architecture implements just four physical cores. In both cases, a desirable scalability of the algorithm when using up to four threads can be appreciated, considering that speedups of roughly 2.5 and 3 are obtained when using four threads for the small and large instances, respectively. Then, the increase in speedup values is reduced when more than four threads are spawned. The explanation is that, in these cases, the processor is using Hyper-Threading since the number of physical cores is only four. Hyper-Threading is a form of simultaneous multithreading technology. A processor with Hyper-Threading consists of two logical processors per core, each of which can execute a specified thread. Unlike two separate physical processors, the logical processors in a Hyper-Threaded core share execution resources (e.g., caches, systems buses). Results show that VS is able to further reduce

execution times even when Hyper-Threading, despite the overhead introduced by the shared resources.

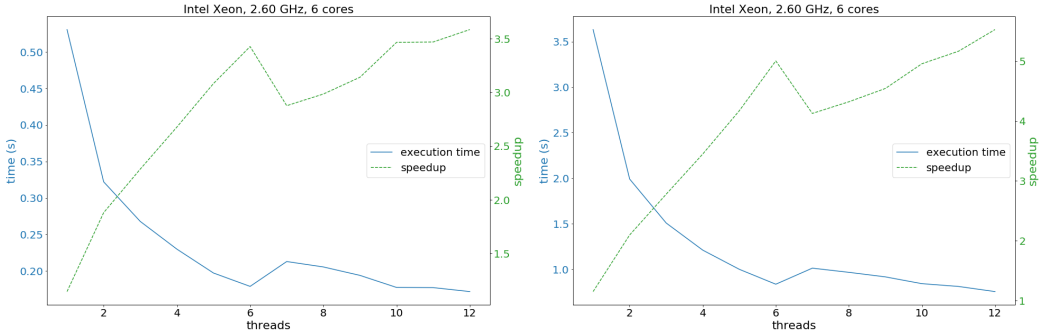
Figure 5.7 presents the average makespan values achieved in the experiment. The trend is that the average makespan values obtained decrease when increasing the number of threads used. Again, the reason is that the higher the number of threads used, the more LSs are executed and, therefore, the higher the chance of computing better results. Consequently, the increase in the quality of the solutions found comes at no cost in the execution time, thanks to the parallel capabilities of the VS paradigm.



(a) 512×16 **(b)** 4096×16

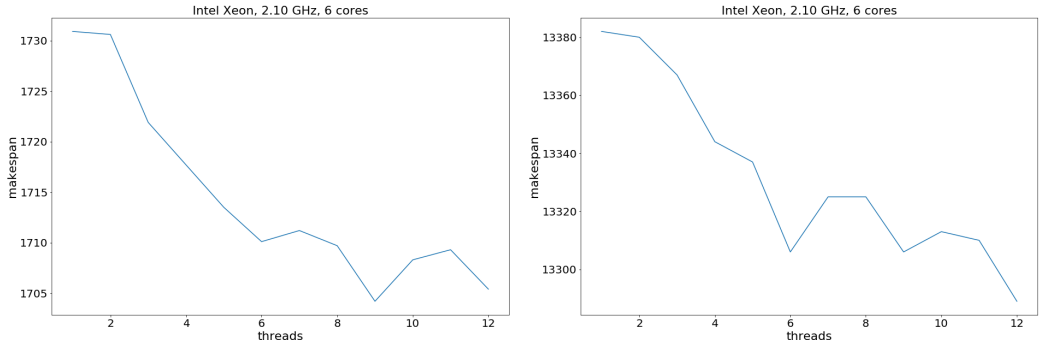
Figure 5.7: Makespan with varying number of threads on Desktop architecture.

5.3.3.3.2 Xeon architecture. Figures 5.8 and 5.9 show the average execution times and speedup values, and the makespan values, respectively, of VS when executing on the Xeon architecture using different number of threads. The results are reported for the two problem instances considered.



(a) 512×16 **(b)** 4096×16

Figure 5.8: Average execution times with varying number of threads on Xeon architecture.



(a) 512×16 **(b)** 4096×16

Figure 5.9: Makespan with varying number of threads on Xeon architecture.

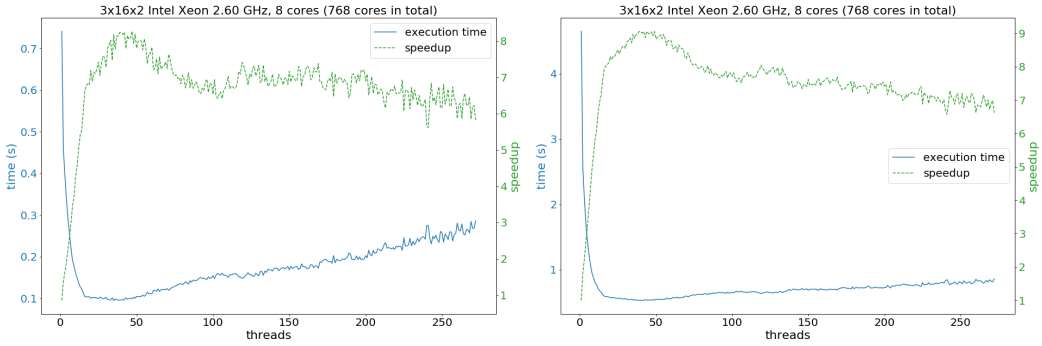
The experiments performed in the Xeon architecture allowed studying the parallel capabilities of VS in a standalone workstation in addition to a regular desktop PC. Speedup values of up to 3.59 and 5.53 were achieved for the 512×16 and 4096×16 instances, respectively. As for the Desktop architecture, results show the good parallel capabilities of the VS paradigm, given that the processor has six physical cores. As in the Desktop case, Hyper-Threading can help in some cases to reduce computation times, but speedup reduces when using this technique. Significant improvements in the quality of the solutions found (i.e., better makespan values) can be computed when increasing the number of threads, without negatively impacting the overall execution time.

5.3.3.3.3 Cluster architecture. The performance of VS was also evaluated on a distributed computing infrastructure. Figure 5.10 shows the execution times and speedup of VS executing on the Cluster architecture when varying the number of processes used. Similarly, Figure 5.11 reports the makespan achieved in this experiment. As in the previous cases, the makespan value decreases (i.e., solutions are better) when increasing the number of processes.

Results show that VS was also able to make efficient use of a distributed computing infrastructure. When comparing the execution times of VS in the Xeon and the Cluster infrastructure, no significant penalties in execution times exist when moving from a shared-memory computing infrastructure to a distributed environment. A slight time execution penalty was observed when comparing the case when only one server is used (threads values 1 to 8 in the plot) to that of using 23 servers (the rightmost results in the plots). The method found the solution in less than 0.4 seconds using 23 servers, in the case

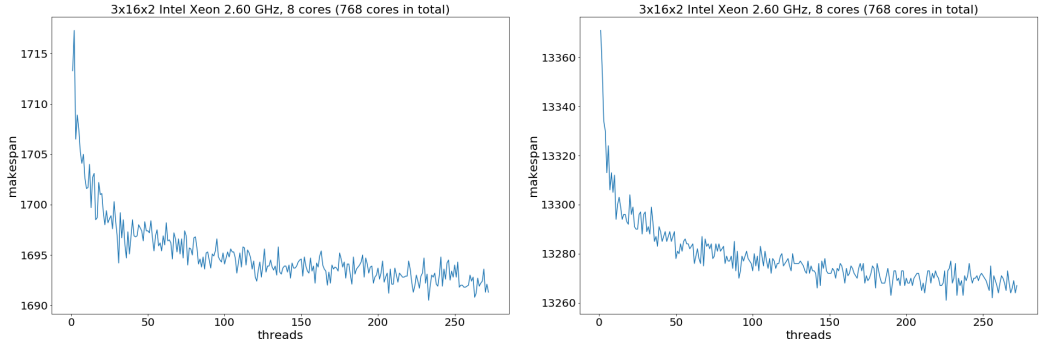
of the small problem instance. A higher computational load in every node is required to achieve better speedup values. This issue is studied in Section 5.3.4, where results for very large problem instances are presented.

Regarding the quality of solutions achieved, a similar trend to the other studied architectures was observed: better makespan values are achieved when increasing the number of computing resources. The presence of spikes in the plots is due to the stochastic nature of the LS. However, the desired downward trend occurred in all studied architectures.



(a) 512×16 **(b)** 4096×16

Figure 5.10: Average execution times with varying number of threads on Cluster architecture.

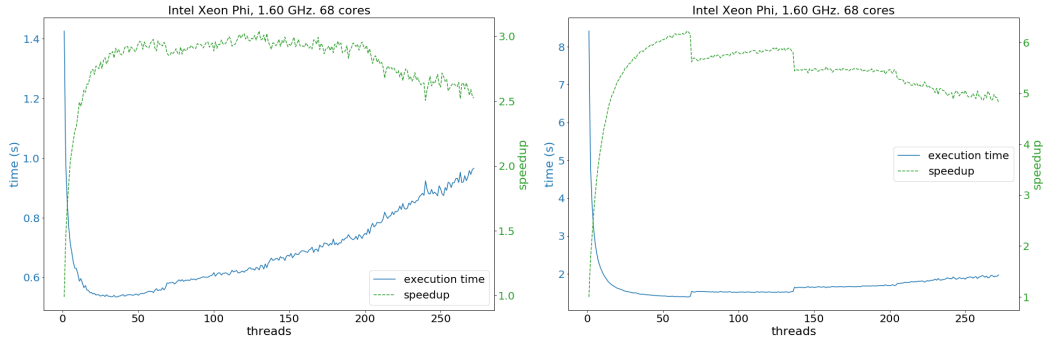


(a) 512×16 **(b)** 4096×16

Figure 5.11: Makespan with varying number of threads on Cluster architecture.

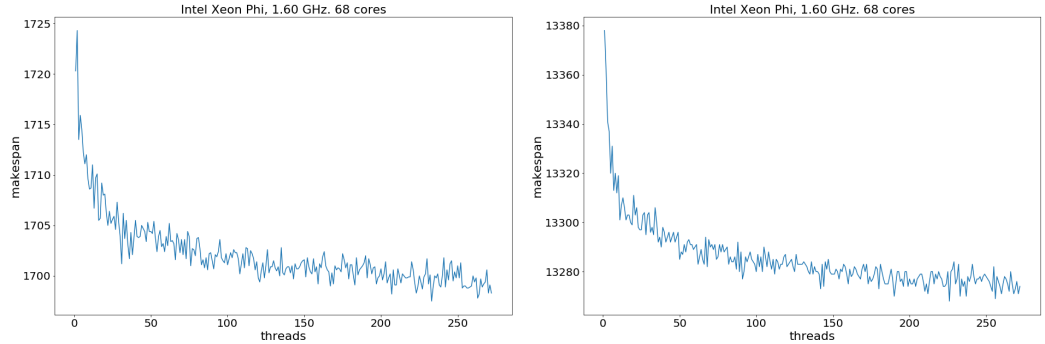
5.3.3.3.4 Phi architecture. The last experiment evaluated the performance of VS on the massively-parallel Phi architecture. Mean execution times with varying number of threads are presented in Figure 5.12, alongside speedup values, while average makespan values are reported in Figure 5.13. Results

show that VS can take advantage of the massive parallel capabilities the Phi architecture offers. Execution times are reduced when increasing the number of threads up to the number of physical cores available, and then slightly increase when more threads are spawned. However, execution times in the Phi architecture are the longest compared to the other studied architectures. The reason is that the computation units in the Phi system are highly limited in resources and speed compared to the other architectures. The processor frequency in the Phi architecture is 1.60 GHz vs. 3.20 GHz in Desktop and 2.60 GHz in Xeon and Cluster architectures. In terms of the quality of the computed solutions, results show that the average makespan is reduced when increasing the number of threads spawn. Thus, VS can take advantage of the availability of more computing resources to further refine the predicted solutions during the improvement phase.



(a) 512×16 **(b)** 4096×16

Figure 5.12: Average execution times with varying number of threads on Phi architecture.



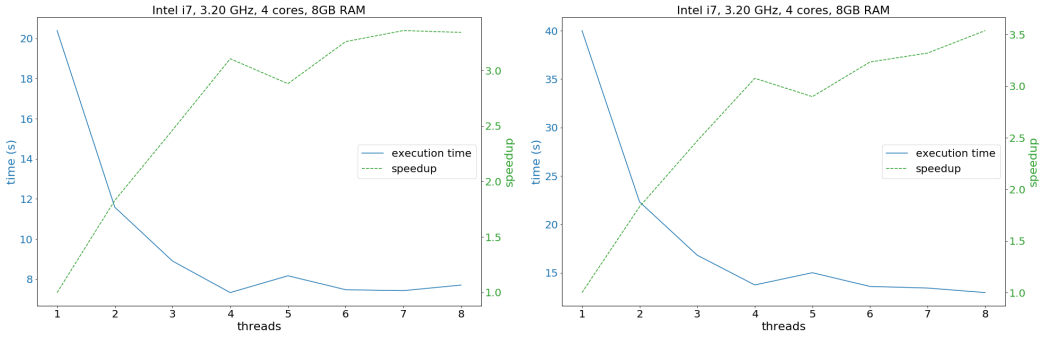
(a) 512×16 **(b)** 4096×16

Figure 5.13: Makespan with varying number of threads on Phi architecture.

5.3.4 Scalability on the problem size

The final experiments on the HCSP studied the performance of the parallel implementation of VS on two very large instances in terms of tasks: 32768×16 and 65536×16. The study on larger problem instances was done on one of the shared-memory architectures (Desktop) and on the distributed-memory architecture (Cluster).

Figure 5.14 shows the results obtained for the Desktop architecture on both problem instances. The number of steps of the LS was increased to one million, given that the complexity of the problem raises with the size of the instance. Figure 5.14 shows that, for the two very large instances, VS exhibits similar behavior to the case of the smaller instances previously studied. Speedup raised almost linearly when increasing the number of cores used, reaching up to 3.4 for four cores. Then, a slight performance loss was noticed when Hyper-Threading was used, but speedup increased again until the whole capacity of the processor was in use, when the best results were obtained.

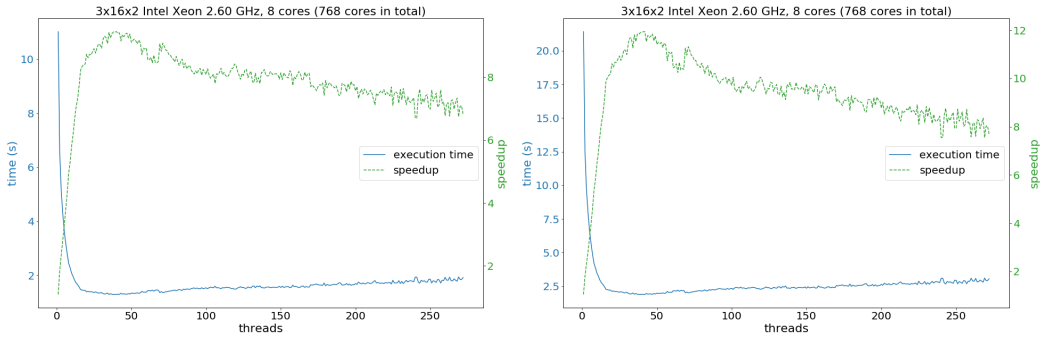


(a) 32768×16 **(b)** 65536×16

Figure 5.14: Average execution times with varying number of threads on Desktop architecture.

The comparison with the MinMin algorithm indicates that it took more than two minutes to find a solution to the largest studied instance (65536×16) in the Desktop architecture. This means that VS, which was trained using MinMin results, was eleven times faster than MinMin on this architecture, and the solution it found was only 3.85% worse than the one reported by MinMin. Execution time is of the essence in heterogeneous computing facilities with online scheduling, as schedulers need to provide quality solutions with reasonable response times. In these cases, faster schedulers may be preferred at the cost of some losses in the computed solutions.

The same problem instances were also solved using the Cluster architecture. Results are presented in Figure 5.15. An improvement in the speedup of VS is appreciated when increasing the problem size. Results show that speedups of up to 12 were achieved for the largest instance in the Cluster architecture. The execution time was around 2.5 seconds, approximately seven times faster than the Desktop architecture. However, thanks to the efficient parallel design of VS, it could be possible to further improve speedup values by assigning a higher workload to the nodes in the cluster, e.g., by implementing a more demanding optimization algorithm for the improvement phase, which might also lead to better results.



(a) 32768×16 **(b)** 65536×16

Figure 5.15: Average execution times with varying number of threads on Cluster architecture.

5.4 Conclusions

This chapter presented the application of VS to solve the HCSP, which corresponds to the first parallel implementation and evaluation of VS. The MPICH and OpenMP libraries were used for the implementation, which allowed efficiently dealing with both shared- and distributed-memory architectures.

The results computed by VS were compared to those computed by the well-known MinMin heuristic, the algorithm used by VS as a reference. Experimental results showed that VS outperformed MinMin in most of 180 problem instances, achieving up to 15% of improvement in terms of makespan. Additionally, VS showed excellent scalability properties when increasing both the computational resources and the problem dimensions.

Then, VS was evaluated on different computing platforms. Experimental evaluation over four different computing infrastructures showed that the massively-parallel design of VS allowed taking advantage of available computing resources to find accurate solutions for the HCSP. Increasing the number of parallel resources helped reducing the execution time of the prediction phase and did not increase the overall execution time, even though the computational demand of VS increases with the number of resources available. Besides, the makespan value (that evaluates the quality of the obtained results) generally decreased (i.e., improves) when increasing the number of threads.

Finally, VS was evaluated when solving very large problem instances—larger than those used during training—comprised of 32 768 and 65 536 tasks and 16 machines. Similar results to those obtained for the smaller instances were found, where the speedup increased with the number of cores, with a loss due to Hyper-Threading when the number of spawned threads exceeded the number of physical cores. However, speedup values were better for the largest instances studied.

Chapter 6

Virtual Savant for the Bus Synchronization Problem

This chapter presents the application of VS to solve the BSP. Section 6.1 introduces the BSP, its mathematical formulation, and a review of related work on the subject. Then, Section 6.2 outlines the application of VS to the BSP. Experimental results are presented and discussed in Section 6.3 and some final remarks are detailed in Section 6.4.

6.1 The Bus Synchronization Problem

This section presents the BSP. A general overview on the subject is presented in Section 6.1.1, the mathematical formulation of the BSP as an optimization problem is outlined in Section 6.1.2, and a brief discussion on the related literature is presented in Section 6.1.3.

6.1.1 Overview

Transportation systems play a major role in urban scenarios and are a fundamental element of modern smart cities (Grava, 2002; Nesmachnow et al., 2017). Public transportation accounts for the majority of trips in large cities and provides the most efficient and environmentally-friendly mean of transportation for citizens. However, the efficacy of public transportation systems requires proper planning of routes, timetabling, buses, drivers, and other relevant subproblems, to provide good quality of service to citizens (Ceder and Wilson, 1986).

Public transportation planners often prefer network topologies comprised of few, short, and densely interconnected bus lines. This design is good from an operational point of view since it provides higher bus frequencies with smaller vehicle fleets. However, passengers are demanded to make more bus transfers instead of traveling directly from their origin to their destination. Passengers dislike transfers. Studies have shown that the perceived time when waiting for a bus or when walking between bus stops can be up to 2.5 times larger than the actual time spent (International Transport Forum, 2014). Consequently, limiting the waiting times experienced by passengers when transferring between buses is a desirable goal from the point of view of citizens.

The BSP consists in finding the headways (i.e., the time between consecutive vehicles) of each bus line in a public transportation network, which allow maximizing the number of synchronized bus transfers. A bus transfer is considered synchronized when the waiting time experienced at the transfer bus stop does not exceed a given threshold. Some public transportation fare schemes impose no limitations on the number of transfers that a passenger can make and consider all bus stops in the network as possible transfer stops, providing flexibility to passengers when planning their routes. In this scenario, the BSP is more complex, as the bus transfer may not be done in the same bus stop and may involve walking between bus stops.

The BSP accounts for the main goals of modern transportation systems: providing a fast and reliable way for the movement of citizens while maintaining reasonable operational costs. Synchronization is one of the most difficult tasks in public transportation planning and has often been addressed intuitively, assuming that experienced operators can take proper decisions (Ceder and Tal, 1999). The mathematical formulation for the BSP, modeled as an optimization problem, is presented next.

6.1.2 Mathematical formulation

The problem model focuses on the quality of service provided to the users, i.e., a better traveling experience with limited waiting times when transferring between buses. This formulation extends the one proposed by Ibarra and Rios (2012) by considering transfers at any pair of bus stops in the public transportation network and accounting for the walking time between bus stops.

The mathematical formulation for the BSP considers the following elements:

- A set of bus lines $I = \{i_1, i_2, \ldots, i_n\}$. For each bus line $i \in I$, $J(i)$ is the set of lines that may synchronize with line i (in a synchronization node, see next item). Buses that operate each line have a maximum capacity of C transfers, i.e., passengers that board the bus in the second leg of a transfer trip.
- A set of synchronization nodes $B = \{b_1, b_2, \ldots, b_m\}$. Each node $b \in B$ is a triplet $<i, j, d_b^{i,j}>$ indicating that lines i (inbound line) and j (outbound line) may synchronize in b, and that the bus stops for lines i and j are separated by a distance $d_b^{i,j}$.
- A planning period $[0, T]$, expressed in time units, and the number of trips of each line i, f_i, needed to fulfill the transfer demand for each line in that period.
- A *traveling time function* $TT : I \times B \rightarrow \mathbf{Z}$. $TT_b^i = TT(i, b)$ indicates the time for buses of line i to reach synchronization node b (measured from the origin of the line). In general, this value depends on several features, including bus type, bus speed, traffic in roads, etc.
- A walking time function $WT : I \times I \times B \rightarrow \mathbf{N}$. $WT_b^{i,j} = WT(i, j, b) = d_b^{i,j}/ws$ indicates the time needed for a pedestrian to walk between bus stops at the synchronization node according to a given walking speed ws.
- A *demand function* $P : I \times I \times B \rightarrow \mathbf{Z}$. $P_b^{i,j} = P(i, j, b)$ indicates the number of passengers that transfer from line i to line j in synchronization node b, within the planning period.
- A maximum waiting time $W_b^{i,j}$ for each synchronization node, indicating the maximum time that passengers are willing to wait for line j, after alighting from line i and walking to the stop of line j, in synchronization node b.
- A valid range of headways, which define the separation (measured in time units), between consecutive trips of the same line. The range of valid headways for bus line i is defined by an interval $[h^i, H^i]$, where values of h^i and H^i are usually enforced by public transportation administrators.

The BSP proposes finding appropriate values for the headways of each bus line to guarantee the best synchronization for all lines with transfer demands in the planning period T. The mathematical model is formulated in Equation 6.1.

The departure time of trip r of bus line i is represented by an integer variable X_r^i. A binary variable $Z_{r,s,b}^{i,j}$ indicates whether trip r of line i and trip s of line j are synchronized or not in node b. The proposed objective function weighs synchronizations according to the number of passengers that transfer in the planning period, thus giving priority to synchronization nodes with larger transfer demands.

$$\text{maximize} \quad \sum_{b\in B}\sum_{i\in I}\sum_{j\in J(i)}\sum_{r=1}^{f_i}\sum_{s=1}^{f_j} Z_{r,s,b}^{i,j} \times \min(\frac{P_b^{i,j}}{f_i}, C) \tag{6.1a}$$

$$\text{subject to} \quad X_1^i \leq H^i \tag{6.1b}$$

$$T - H^i \leq X_{f_i}^i \leq T \tag{6.1c}$$

$$h^i \leq X_{r+1}^i - X_r^i \leq H^i \tag{6.1d}$$

$$(X_s^j + TT_b^j) - (X_r^i + TT_b^i) > WT_b^{i,j} \text{ if } Z_{r,s,b}^{i,j} = 1 \tag{6.1e}$$

$$(X_s^j + TT_b^j) - (X_r^i + TT_b^i) \leq W_b^{i,j} + WT_b^{i,j} \text{ if } Z_{r,s,b}^{i,j} = 1 \tag{6.1f}$$

$$X_r^i - X_{r-1}^i = X_s^i - X_{s-1}^i \forall r, s, r > 1, s > 1 \tag{6.1g}$$

$$X_r^i \in \{0, \ldots, T\}, Z_{r,s,b}^{i,j} \in \{0, 1\} \tag{6.1h}$$

The objective function of the problem (Equation 6.1a) proposes maximizing the number of synchronized transfers, weighed by the corresponding transfer demand for each trip in each synchronization node. When computing the objective function, the demand is split uniformly among the f_j trips of line j. This is a realistic assumption for planning periods where demand does not vary significantly. The number of passengers considered on each synchronization node is bounded by the transfer capacity for buses C. Equations 6.1b–6.1h specify the constraints of the problem.

Equation 6.1b states that the first trip of each line must start before the upper bound for headways for that line. A similar constraint in Equation 6.1c is included for the last trip of each line, which must end before the planning period T, thus, it must start after the end of the period minus the upper bound for headways for that line. The constraint in Equation 6.1d guarantees that the computed headways of each line are bounded to the range of valid headways. Equations 6.1e and 6.1f define the condition for two trips to be synchronized at a given node: trip r of line i and trip s of line j are synchronized at node b if passengers are able to transfer, considering the time needed to walk between

the bus stops in the node $WT_b^{i,j}$ and the maximum time that users are willing to wait for the second bus in the transfer to arrive, $W_b^{i,j}$. Equation 6.1g states that all headways for a given bus line are constant. This constraint is consistent with the assumption of uniformly distributed transfer demands within the planning period. However, the proposed problem formulation allows relaxing this constraint to model a more complex variant of the BSP, where bus headways of a given line may change within the planning period. This is a distinction of the proposed model, considering that related works usually split the problem in multiperiod timetabling planning, forcing to solve several chained problems with fixed headways (Ibarra et al., 2016). Finally, Equation 6.1h defines the domain for the decision variables of the problem.

6.1.3 Related work

Daduna and Voß (1995) studied the timetable synchronization problem on bus networks to minimize the waiting time of passengers. Different objectives were formulated, including a weighted sum considering transfers and the maximum waiting time while transferring. Simulated Annealing and Tabu Search were analyzed for simple versions of the problem. Tabu Search computed better solutions than Simulated Annealing over randomly generated examples based on the Berlin Underground network. Besides, three real-world cases from different German cities were studied. The trade-off between operational costs and user efficiency suggested that multiobjective approaches should be considered.

Ceder et al. (2001) studied the problem of maximizing the number of synchronization events between bus lines at shared stops, i.e., maximizing the number of simultaneous arrivals. A heuristic greedy approach that selects nodes from the bus network was proposed to efficiently solve the problem by defining custom timetables. The work focused on simultaneous bus arrivals and the reported results consisted of examples that illustrated synchronizations on small instances with few nodes and few bus lines.

Fleurent et al. (2004) considered a synchronization metric including weights defined by experts and public transportation authorities to minimize vehicle costs. A heuristic algorithm was proposed to solve network flow problems that account for the synchronization metric and other operation costs. Experiments performed on just two small scenarios from Montréal, Canada, computed different timetables when varying the weights used in the proposed metric.

Ibarra and Rios (2012) studied the bus synchronization problem in the bus network of Monterrey, Mexico. A flexible formulation of the problem was proposed, considering a time window between travel times to account for traffic congestion and other situations. A Multi-start Iterated Local Search (MILS) was evaluated over eight instances modeling the bus network in Monterrey (15 to 200 bus lines and 3 to 40 synchronization points). MILS was compared against a Branch & Bound method (which failed to compute optimal solutions in two hours) and a simple upper bound computed by adding the possible trips to synchronize. The method was able to compute efficient solutions for medium-sized instances in less than one minute, but the gaps of MILS did not scale properly. Later, Ibarra et al. (2016) solved the multiperiod bus synchronization problem, optimizing multiple trips of a given set of bus lines. MILS, Variable Neighborhood Search, and a simple population-based approach were proposed to solve the problem. All methods computed solutions with similar quality to an exact approach over synthetic instances with few synchronization points. Multiperiod timetables were up to 20% better than merging single period timetables. Results for a sample case study using data for a single bus line showed that maximizing synchronizations for a specific node usually reduces synchronizations for other nodes.

Our conference article presented the BSP formulation outlined in Section 6.1.2 and an EA to solve it (Nesmachnow et al., 2020) . The problem formulation extended the one proposed by Ibarra and Rios (2012) by considering scenarios where every pair of bus stops are possible transfer nodes to synchronize. Candidate solutions to the problem were modeled in the EA using integer vectors, where each value represented the headway (in minutes) of a bus line. The initial population of the EA was comprised of randomly generated solutions that satisfied the problem constraints as well as seed solutions corresponding to the current reality defined by the city authorities of the studied scenario and from greedy approaches for the problem. Evolutionary operators included a traditional tournament selection, a two-point crossover operator, and a Gaussian mutation which modified the headway of the lines. The fitness function accounted for the number of synchronized trips and their corresponding demands, according to the problem formulation. Problem instances based on real data from the public transportation system in Montevideo, Uruguay were used for the experimental evaluation. Results showed that the proposed evolutionary approach computed accurate solutions, improving up to 13.33%

in the fitness values and up to 24.20% in the waiting times, when compared to the current real timetable in Montevideo. This EA is used as the reference algorithm for the VS implementation for BSP which is presented next.

6.2 VS for the BSP

Due to its flexible design, VS is agnostic in terms of which machine learning classifier is used. In order to show this flexibility, the VS design for the BSP uses RF as a classifier for the prediction phase, in contrast with the previous implementations of VS which used SVMs. RF and SVM were compared in preliminary experiments for the BSP and RF achieved a higher accuracy.

The process of generating the training set for the BSP is outlined in Figure 6.1. RFs are trained using the solutions computed by an EA used as a reference (Nesmachnow et al., 2020). The best solution found on each independent execution of the reference EA over each training instance is used to build the dataset of solved BSP instances.

Each synchronization node in a given instance is independently considered in the learning process of VS. Different combinations of features can be considered during training. In fact, different combinations were analyzed and the experimental results are reported in Section 6.3.3.1. The training set generation process depicted in Figure 6.1 is described using the combination of features that achieved the best results in these experiments, where each feature vector is comprised of:

- *Synchronization node features*: walking time between the pair of bus stops, transfer demand, and maximum waiting time.
- *Inbound line features*: travel time to synchronization node, minimum allowed headway, and maximum allowed headway.
- *Outbound line features*: travel time to synchronization node, minimum allowed headway, and maximum allowed headway.

Two different classifiers are trained: one to predict the headway of the inbound line (y_b^i) and another one for the headway of the outbound line (y_b^j). Thus, two different training datasets are built, each comprised of the same number of training vectors with identical features but with different labels, corresponding to the headways of the inbound and outbound lines as computed by the reference EA. In summary, one BSP instance yields as many

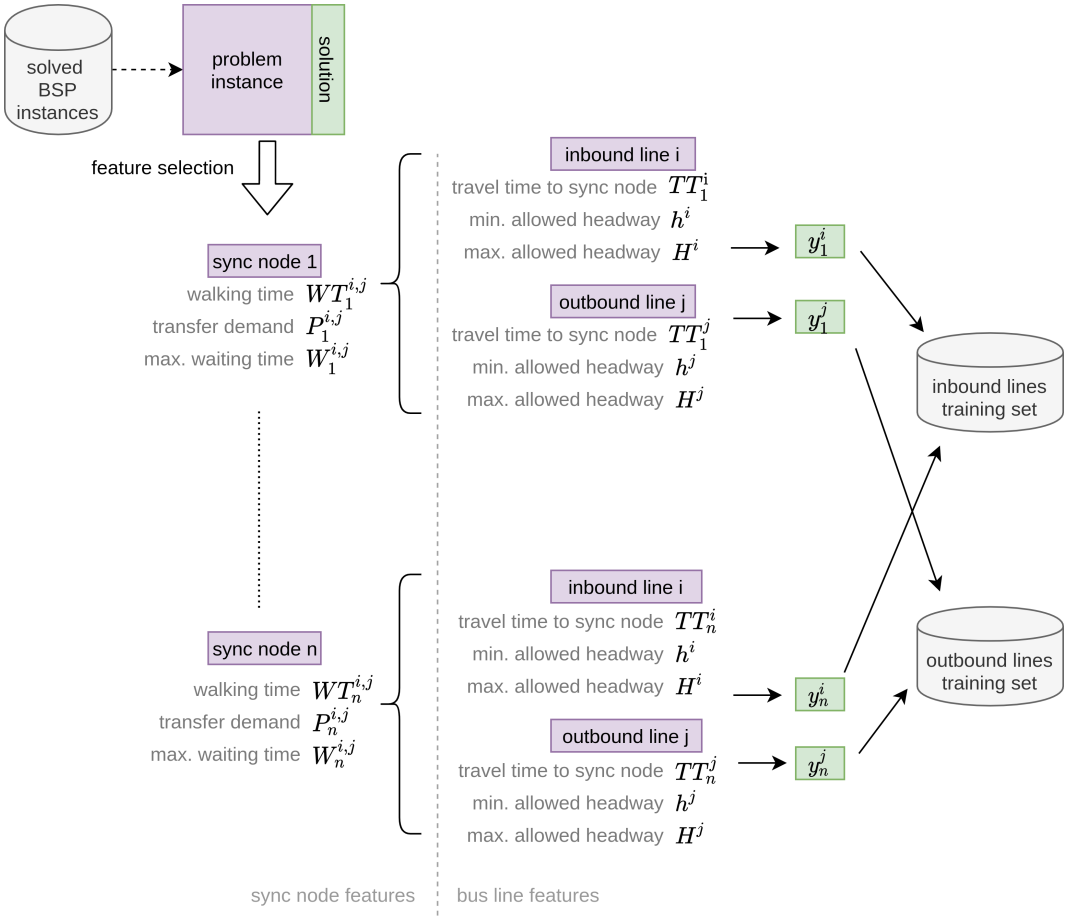


Figure 6.1: VS training set generation for the BSP.

training observations to each dataset as the number of synchronization nodes in the instance times the number of independent executions of the reference EA. The scikit-learn library (presented in Section 2.4.3.1) was used for the implementation of VS with RF as classifiers.

The workflow of VS when solving the BSP is presented in Figure 6.2. VS receives as input the BSP instance, including the features corresponding to the synchronization nodes and bus lines. Since the training phase considers each synchronization node independently, the prediction can be highly parallelized. The prediction for each synchronization node can be performed in parallel using copies of the two trained classifiers. Moreover, the prediction of the headways of the inbound and outbound lines for a given synchronization node can also be parallelized, since they are performed by different classifiers. The output of each RF classifier is the predicted headway of the inbound or outbound line of the synchronization node ($\hat{y_b^i}$ and $\hat{y_b^j}$ in Figure 6.2).

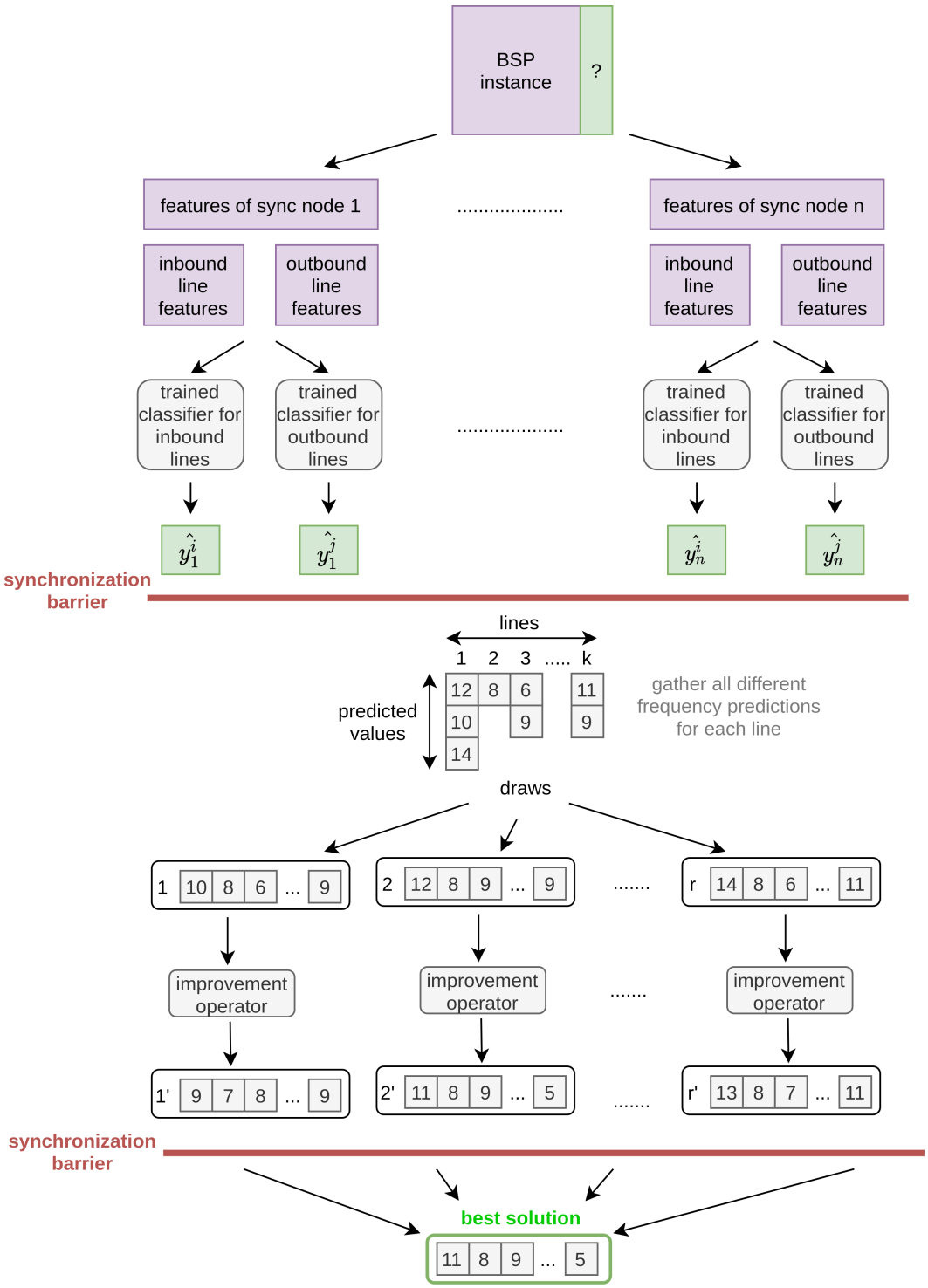


Figure 6.2: VS workflow for the BSP.

A given bus line can be part of multiple synchronization nodes, acting as either the inbound or outbound line. Thus, when the predictions of each classifier are gathered, multiple—possibly conflicting—predictions can be made for a given bus line. Therefore, predictions are gathered forming a list for each bus line with the different predicted headways. Additionally, a prediction may be invalid if the predicted headway is not within the range of allowed headways for that line (defined by h^i and H^i). This is a possible scenario since the set of classification labels is comprised of every allowed headway for every bus line in the training set. Thus, a classification error could lead to predicting a label that corresponds to a valid headway for some line in the training set but not for the bus line being predicted. Consequently, predicted headways are checked in this step and unfeasible predictions are not included in the list of candidate headways for the line. Once all the predictions are gathered, multiple candidate solutions are generated to be refined through the improvement operator. Candidate solutions are built using the list of headways computed in the prediction phase and according to the following criteria. For a given bus line:

- if only one valid prediction is available for the line, the corresponding headway is fixed for the line in the candidate solution.
- if more than one valid prediction is available for the line, a random headway is selected among the set of valid predictions (according to a uniform probability distribution).
- if no valid predictions are available for the line, a random headway is chosen within the valid range of headways for that line (according to a uniform probability distribution). This extreme case can happen when all the predictions for a given bus line correspond to headways outside the valid range of headways for the line.

Following this procedure, multiple candidate solutions are generated to be refined. The refinement of solutions can also be made in parallel by generating and improving solutions using a master-slave approach. The improvement operator consists of a simple LS that selects a bus line in each step and randomly changes its assigned headway according to a uniform distribution in the range of valid headways for the line. The change is accepted if the quality of the solution improves and is discarded otherwise. The quality of the solution is measured through a score function, which reflects the problem formulation,

and is used by the reference EA as a fitness function. The score function accounts for the number of synchronized transfers and their corresponding demands. The score of a given solution is computed by the procedure described in Algorithm 4, which is described next.

Algorithm 4: Score function to evaluate BSP solutions.

```
   input  : solution, instance
   output: score
1  T ← planning_period(instance)
2  C ← bus_capacity(instance)
3  score ← 0
4  foreach node b in get_sync_nodes(instance) do
5      TT_b^i, TT_b^j, WT_b^{i,j}, P_b^{i,j}, W_b^{i,j} ← get_features(b)
6      y_b^i, y_b^j ← get_headways(solution, b)
7      for m=0 to T step y_b^i do // Iterate over inbound trips
8          for n=0 to T step y_b^j do // Iterate over outbound trips
9              wait_time = (n + TT_b^j) − (m + TT_b^i) − WT_b^{i,j}
10             if wait_time > y_b^j then
                   // At most, passengers wait for the full headway
                   wait_time ← y_b^j
11             end
12             if wait_time > 0 & wait_time ≤ W_b^{i,j} then
                   // objective function is multiplied by T
                   // to avoid the division in f_i = T/y_b^i
13                 score ← score + min(P_b^{i,j} · y_b^i, C · T)
14                 break
15             end
16         end
17     end
18 end
```

The planning period T and the capacity for transfers C are global information obtained from the problem instance (lines 1- 2). The score function iterates over each synchronization node in the instance (loop in line 4), obtaining all the features of the synchronization node from the problem instance (line 5), i.e., travel time for the inbound and outbound lines (TT_b^i, TT_b^j), walking time between bus stops ($WT_b^{i,j}$), passenger demand ($P_b^{i,j}$), and maximum allowed waiting time ($W_b^{i,j}$). Additionally, the headway for the inbound and outbound lines (y_b^i and y_b^j) are retrieved from the solution being evaluated (line 6). The algorithm then iterates over each pair of trips of the inbound and outbound

lines (loops in line 7 and 8). The waiting time for a pair of trips is computed as the difference between the time the outbound and the inbound lines arrive at the synchronization node, subtracting the walking time between bus stops (line 9). In the worst case, passengers have to wait for the full headway of the outbound line (lines 10-11). A pair of trips is synchronized if the computed waiting time is below the threshold $W_b^{i,j}$ that passengers are willing to wait for the outbound line (line 12). In this case, the demand is accumulated according to the problem formulation (line 13). The objective function is multiplied by the planning period T to avoid the division present in $f_i = T/y_b^i$ in the original formulation. This simple transformation was included to be able to solve small problem instances to optimality using integer linear programming solvers, which is regarded as a line of future work. The break directive in line 14 ensures that each trip of the inbound line is synchronized with one trip of the outbound line at most.

Once all candidate solutions are improved, results are gathered and the best computed solution is returned.

6.3 Experimental evaluation

This section presents and discusses the experimental results of VS when solving the BSP. First, the problem instances used for the experiments are described in Section 6.3.1. Then, the baseline algorithms used for comparison are presented in Section 6.3.2. The experimental results over a set of synthetic problem instances are outlined in Section 6.3.3 and the results corresponding to realistic instances from the public transportation network of Montevideo, Uruguay, are presented in Section 6.3.4.

6.3.1 Problem instances

Two sets of problem instances were considered: synthetic instances and realistic instances corresponding to the public transportation system of Montevideo, Uruguay. The methodologies for building both sets of instances are described next.

6.3.1.1 Synthetic BSP instances

A set of 130 synthetic instances for the BSP was built using a grid topology that models the streets in a city. The process to generate each problem instance is described next.

First, a grid of size $M \times N$ blocks was defined, on which the bus network was designed. Bus lines were randomly generated considering two different types: i) *regular lines*, with starting and ending points far apart; and ii) *circular lines*, with the start and end of the line close to each other. This two types of bus lines model characteristics usually present in the design of real bus networks. L lines were generated, $\alpha \cdot L$ were regular lines and $(1-\alpha) \cdot L$ were circular lines, with $\alpha \in [0, 1]$. Once the bus lines were defined, bus stops for each line were placed at intersections of the grid, considering a fixed distance d^* between bus stops of the same line. A set of candidate synchronization nodes was selected over the defined bus network, considering those lines with nearby bus stops. Two bus stops s_1 and s_2 must be at a distance $d(s_1, s_2) \leq r$ (measured in blocks) to be considered as a candidate synchronization node. Out of the S candidate synchronization nodes, subsets of size $\beta \cdot S, \beta \in [0, 1]$ were randomly selected to define different problem instances using the same network topology. The walking time between bus stops $WT_b^{i,j}$ is expressed in minutes and is computed using the Manhattan distance, assuming a walking speed $ws = 1$ (measured in blocks per minute). The travel times of each bus were defined assuming a constant speed s, expressed in blocks per minute. For each line, the minimum allowed headway h^i was randomly selected with uniform probability from $[h_{\min}, h_{\max}]$ with $h^i \in \mathbb{N}$. The maximum allowed headway H^i was defined as $H^i = \lceil c \cdot h^i \rceil$, with c randomly selected with uniform probability from $[c_{\min}, c_{\max}]$. The demand $P_b^{i,j} \in \mathbb{N}_0$ for each synchronization node was selected at random from $[P_{\min}, P_{\max}]$ with uniform probability. The maximum allowed waiting time for each transfer was defined according to the maximum allowed headway of the outbound line: $WT_b^{i,j} = \lambda \cdot H^j$. Smaller values of λ correspond to instances with a tighter time constraint, modeling passengers unwilling to wait for the bus for long periods. The set of problem instances were defined according to the parameters outlined in Table 6.1.

Table 6.1: Parameter configuration for synthetic BSP instances.

parameters	*values*
T	120 minutes
C	5
$M \times N$	$(50 \times 50), (300 \times 300), (350 \times 350), (400 \times 400), (450 \times 450)$
L	$50, 300, 350, 400, 450$
β	0.5
α	0.5
d^*	3 blocks
r	2 blocks
ws	1 block per minute
s	4 blocks per minute
$(h_{\min}, h_{\max})$	$(5, 15)$ minutes
$(c_{\min}, c_{\max})$	$(3, 5)$
$(P_{\min}, P_{\max})$	$(6, 40)$
λ	$0.7, 0.9, 1.0$

6.3.1.2 Realistic BSP instances

Realistic BSP instances were built using data from the public transportation system in Montevideo, Uruguay. Different sources of data were combined to generate the problem instances including open data and ticket sales data from smartcard transactions. Open data from the city transportation authorities included bus lines, bus stops, and timetables for the entire public transportation system. Transfers information corresponded to real data from ticket sales in 2015 using smartcards (Massobrio, 2018). Smartcards are compulsory for passengers transferring between bus lines: if paying using cash, passengers must purchase two separate tickets. Since smartcards are required for transfers, it is fair to assume that the transfer records of passengers using smart cards are a good estimator of the behavior of the complete universe of passengers in the system. The key elements of the generated instances are described next.

A planning period of $T = 120$ minutes was used, considering bus trips on working days departing between 12.00 and 14.00, when the public transportation system is at peak usage in terms of ticket sales (Massobrio and Nesmachnow, 2020). A set of 45 problem instances was defined, 15 for each of the three different dimensions considered (30, 70, and 110 synchronization nodes). Synchronization nodes were randomly chosen with uniform probability among the 170 most demanded transfers for the considered period, which

are depicted in Figure 6.3. The demand on each synchronization node $P_b^{i,j}$ was assigned according to the real transfer demand in May 2015, as registered in the smartcard transactions. The available transfer capacity C on each synchronization node was fixed to $C = 5$ for all problem instances. This value was set based on real data from ticket sales in 2015 (Massobrio, 2018) and accounts for other passengers that may be already on the bus as well as passengers directly boarding the bus (without transferring from another line). A preliminary sensitivity analysis was performed that showed that the computed solutions were weakly dependent on the chosen value of C for the studied instances. The walking time between bus stops $WT_b^{i,j}$ was computed according to the distance between bus stops and assuming a constant walking speed of 6 km/h. The travel time of the bus lines to the synchronization node TT_b^i and TT_b^j were defined using the publicly-available timetables, averaging the travel times of all trips of each line within the considered planning period. The range of allowed headways for each line $[h^i, H^i]$ was also computed based on the public timetables, considering all trips within the planning period. The maximum waiting time for each transfer was defined according to the maximum headway for the outbound line of the transfer, i.e., $WT_b^{i,j} = \lambda \cdot H^j$, with $\lambda \in \{0.7, 0.9, 1.0\}$.

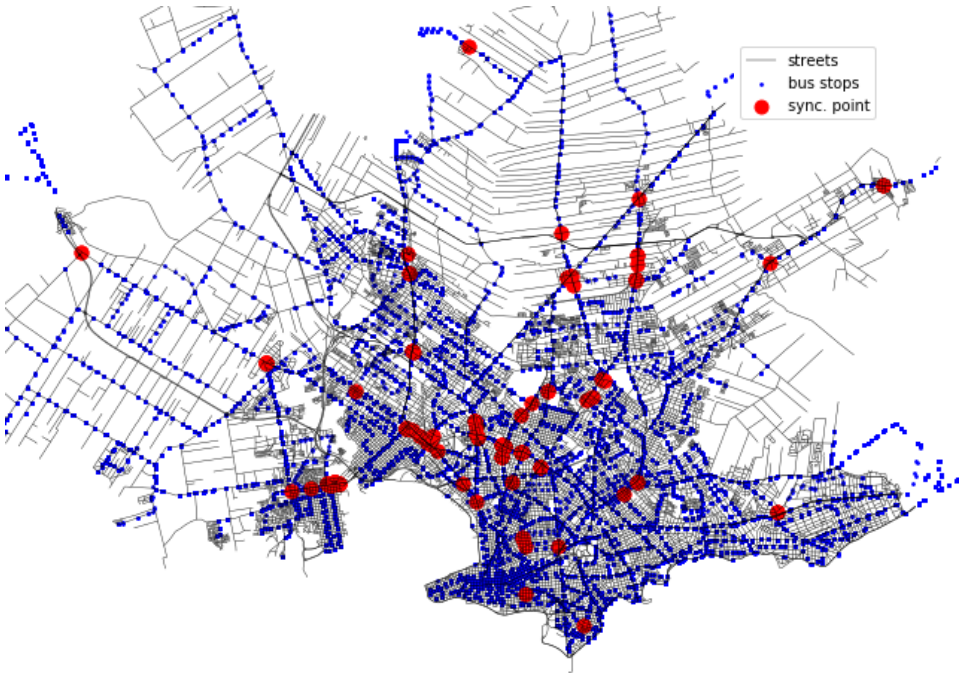


Figure 6.3: Location of synchronization nodes in realistic BSP instances in Montevideo, Uruguay.

6.3.2 Baseline solutions for results comparison

A set of baseline solutions were considered to evaluate the efficacy of VS, which are presented next.

- *EA*: the best solution computed by the reference EA.
- h^i: a solution where the headway of each bus line is assigned to its minimum allowed value. This solution is too expensive to put into practice, since configuring all lines to operate at minimum headways accounts for a very large number of vehicles and operating costs.
- *random*: a randomly generated solution where the headway of each bus line is chosen within the valid range of headways for that line (according to a uniform probability distribution).
- *LS (1000)*: a *random* solution followed by 1000 steps of the same LS operator used in the improvement phase of VS.
- *real*: the current solution according to the public timetable available for the transportation system of Montevideo, Uruguay, only used in the experimental evaluation of realistic BSP instances. This solution was computed by assigning to each bus line the average headway determined by the city authorities for the considered planning period.

6.3.3 Results on synthetic instances

This section outlines the experimental evaluation of VS over the set of synthetic BSP instances.

6.3.3.1 Training of VS

The set of 130 problem instances described in Section 6.3.1.1 was divided into two subsets: a training set comprised of 70 instances and a test set comprised of 60 instances, including the larger instances in terms of the number of bus lines. Thirty independent executions of the reference EA were performed over each instance of the training set and the best solution found on each execution was included to generate the dataset of solved instances, as described in Section 6.2.

Several feature configurations were evaluated and compared in terms of accuracy (i.e., the number of correct predictions out of all predictions made). The first configuration (C1) considers the features corresponding to the synchronization node for the feature vector of the classifiers. More precisely, con-

figuration (C1) holds the bus lines identifiers i and j, the walking time between bus stops $WT_b^{i,j}$, the transfer demand $P_b^{i,j}$ for the node, the maximum allowed waiting time $W_b^{i,j}$, and the traveling times of both bus lines to the synchronization node (TT_b^i and TT_b^j). The second configuration (C2) corresponds to the same feature vector as C1 but without the bus line identifiers. Configuration C3 consists of adding three global features from the problem instance to configuration C2: the total number of bus lines L, the total number of synchronization nodes S, and the planning period T. Finally, configuration C4 is comprised of the same features as C2 but including also the minimum and maximum allowed headways for the inbound and outbound bus lines. Table 6.2 outlines the different feature configurations considered and their corresponding accuracy.

Table 6.2: Accuracy of different feature configurations for synthetic BSP instances.

conf.	*feature vector*	*accuracy*	
		inbound	*outbound*
C1	$\langle i, j, WT_b^{i,j}, P_b^{i,j}, W_b^{i,j}, TT_b^i, TT_b^j \rangle$	0.17	0.31
C2	$\langle WT_b^{i,j}, P_b^{i,j}, W_b^{i,j}, TT_b^i, TT_b^j \rangle$	0.17	0.29
C3	$\langle L, S, T, WT_b^{i,j}, P_b^{i,j}, W_b^{i,j}, TT_b^i, TT_b^j \rangle$	0.17	0.29
C4	$\langle h^i, H^i, h^j, H^j, WT_b^{i,j}, P_b^{i,j}, W_b^{i,j}, TT_b^i, TT_b^j \rangle$	0.50	0.48

The similar results of configuration C1 and C2 suggest that the bus line identifiers are not necessary for accurately predicting headways. Similarly, results achieved using C3 suggest that the accuracy does not improve when adding global information from the problem instance. Configuration C4 significantly outperformed all the other feature configurations, achieving accuracy values of 0.50 and 0.48 for the inbound and outbound lines, respectively. Consequently, configuration C4 was used for the remainder of the experimental evaluation.

6.3.3.2 Accuracy per instance

The prediction phase of VS was evaluated in terms of accuracy. Figure 6.4 presents a bar plot of the prediction accuracy for each of the 60 instances in the test set of BSP synthetic instances. Blue bars correspond to the accuracy achieved by the classifier for inbound lines and orange bars correspond to the accuracy of the classifier for outbound lines.

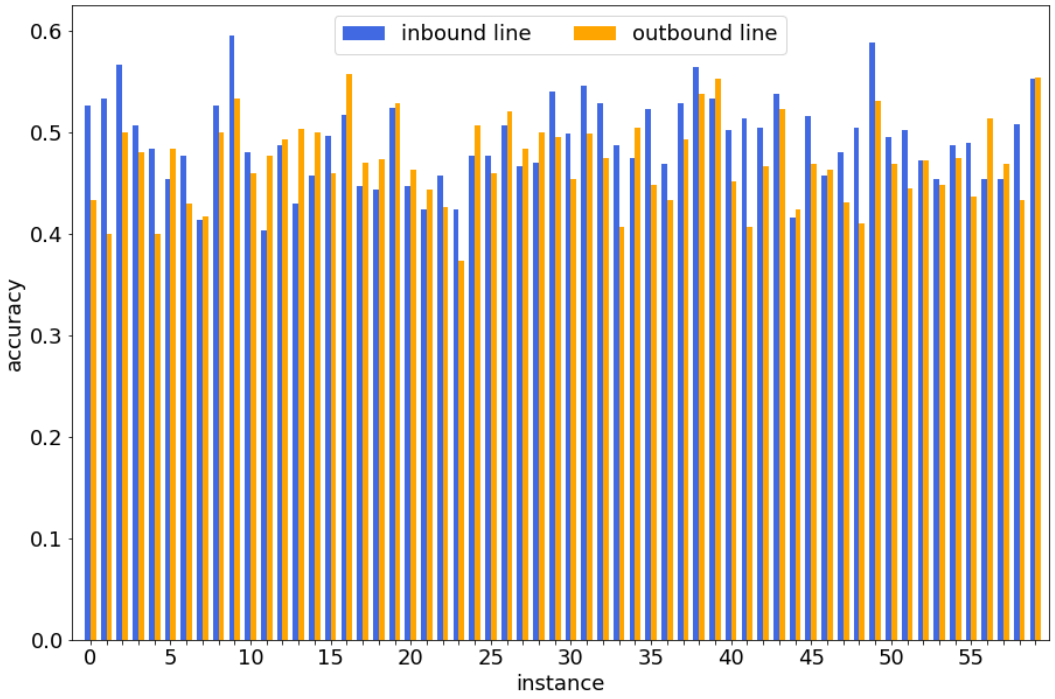


Figure 6.4: Prediction accuracy per instance for synthetic BSP instances.

The classifier for predicting the headways of inbound lines has 59 classes, i.e., it must predict one headway value from within a set of 59 possible values. The classifier for outbound lines has 68 possible classes. Thus, a random prediction over the possible values would render an average accuracy of 0.017 for the inbound classifier and 0.015 for the outbound classifier. Taking this into account, results in Figure 6.4 show a very good prediction accuracy across all problem instances for both classifiers. The maximum prediction accuracy was 0.60 and 0.56 for the inbound and outbound classifiers, respectively. In median over all problem instances, the inbound classifier achieved an accuracy of 0.49 while the outbound classifier achieved an average accuracy of 0.47.

6.3.3.3 Number of valid predictions

Since predictions are independently made for each synchronization node, many different valid predictions can be made for the same line, acting as either the inbound or outbound line of multiple synchronization nodes. Additionally, predictions may not fall within the allowed range of headways for the line, and thus some lines may be left without a valid headway in the prediction phase.

As explained in Section 6.2, the process of generating candidate solutions

depends on the number of valid predictions for each line. With zero valid predictions the headway is randomly set within the allowed range, and when more than one valid prediction is available one is picked at random to generate the candidate solution. Thus, it is important to study the number of valid predictions per line for the studied instances. Figure 6.5 shows a histogram of the number of valid predictions accumulated over all studied instances. Disaggregated histograms for each instance can be found in Figures A.1 and A.2 in Appendix A.

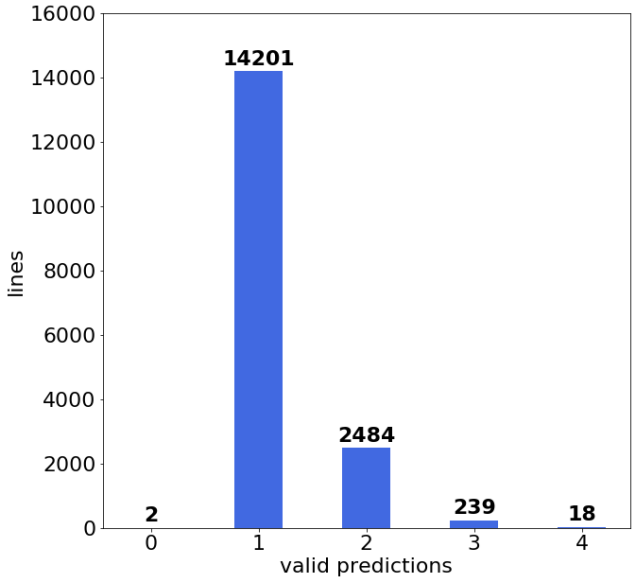


Figure 6.5: Number of valid predictions per line for synthetic BSP instances.

A single headway value was predicted for most lines, which was used directly when generating candidate solutions. No valid prediction was made for just two bus lines, considering all studied instances. For each of these bus lines, the headway value was randomly selected according to a uniform probability distribution defined within the allowed range of headway values for the line. In some cases, more than one valid headway was predicted for the same line. At most four values were predicted for the same bus line. In these cases, one of the predicted values was selected at random (according to a uniform distribution) during the generation of candidate solutions.

6.3.3.4 Comparison with baseline solutions

VS was compared against the baseline solutions described in Section 6.3.2. Figure 6.6 shows the best scores achieved by VS and the baseline solutions.

Score values are computed as defined in Algorithm 4 and normalized using the results computed by the EA as reference. Results correspond to 30 independent executions of each algorithm on each problem instance, except for h^i, which is deterministic. Results for three different configurations of VS are displayed: VS (0), corresponds to the prediction phase of VS (i.e., without applying the improvement operator), VS (1000) and VS (5000) correspond to VS with a 1000-step and 5000-step LS improvement operator, respectively. Numerical results for each instance are reported in Table A.1 and raw results (without normalizing) are outlined in Table A.2, both in Appendix A. Moreover, Figures A.3–A.6 in Appendix A present boxplots comparing the results achieved with the different configurations of VS against the results computed by the reference EA. Boxplots considering all baseline solutions can be found in Figures A.7–A.10 also in Appendix A.

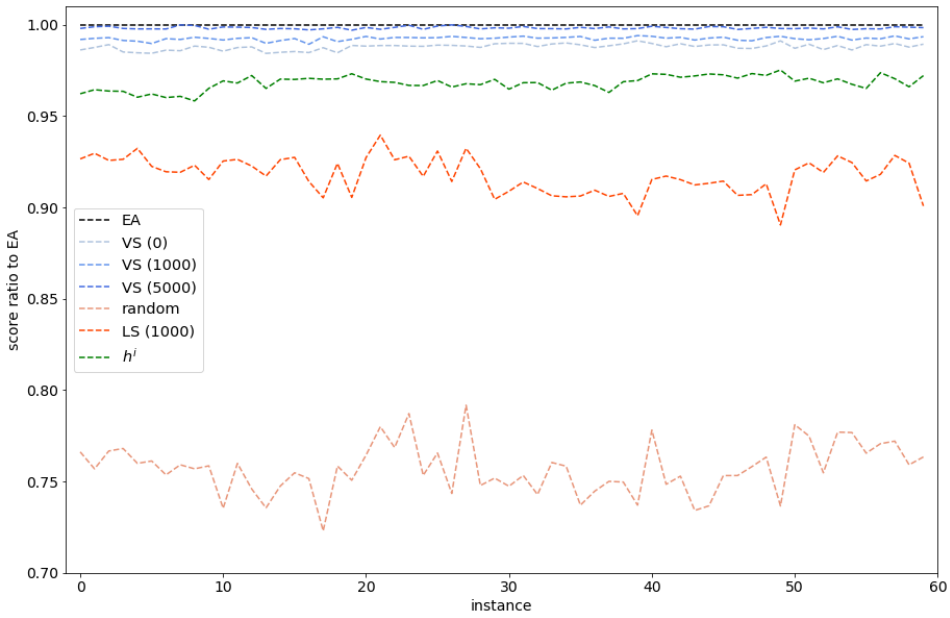


Figure 6.6: Normalized results comparison of VS with baseline solutions for synthetic BSP instances.

Results show that VS computed accurate results even when considering only the prediction phase. VS (0) outperformed the randomly generated solution by up to 26.4% and the minimum headway solution by up to 3.0%. The prediction phase of VS computes solutions that are up to 99.1% as good as the reference EA (98.8% in median and 98.4% in the worst case). These results show that the prediction phase of VS is enough to compute highly-accurate solutions for the evaluated instances.

Including the LS improvement operator of VS allowed further refining the computed solutions. When adding a 1000-step LS improvement operator, solutions computed by VS were up to 99.4% as good as the reference EA (99.3% in median and 98.9% in the worst case). With an LS of 5000 steps, VS computed solutions 99.9% as good as the reference EA in the best case (99.8% in median and 99.7% in the worst case). The effectiveness of the LS operator is demonstrated when comparing the results of the random solution with the results of LS (1000), which refines the random solution with 1000 steps of the LS. Results show that the LS allowed improving the random solution by up to 19%. Thus, each phase of VS contributes to the overall computed result.

VS was trained using instances generated over grids of up to 400×400 and evaluated on instances corresponding to grids of size 450×450. Results show that VS was able to accurately scale in the problem dimension, computing high-quality solutions in instances larger than those seen during training.

6.3.4 Results on realistic instances

This section outlines the experimental evaluation of VS over the realistic BSP instances from the public transportation network in Montevideo, Uruguay.

6.3.4.1 Training of VS

The set of instances described in Section 6.3.1.2 was divided into a training set, comprised of the 30 instances of smaller size (30 and 70 synchronization nodes) and a test set comprised of 15 instances with 110 synchronization nodes. The training set includes the best solution found on each of the 30 independent executions of the reference EA. Thus, the training sets for the inbound and outbound lines each had 45 000 vectors, corresponding to the 30 independent executions of each of the 15 instances of sizes 30 and 70 synchronization nodes ($30 \times 15 \times (30 + 70) = 45000$ vectors).

The features included in the training vector were those that achieved the best results in the analysis described in Section 6.3.3.1 for the synthetic instances, i.e., the range of allowed headways and travel times for both lines in the synchronization node, the walking time between the bus stops, the maximum allowed waiting time, and the passenger demand. As with the synthetic BSP instances, two separate RF classifiers were trained: one to predict the headways of inbound lines and one for the outbound lines.

6.3.4.2 Accuracy per instance

Figure 6.7 outlines the prediction accuracy of each classifier for each instance in the dataset. Blue bars correspond to the inbound line headway predictions while orange bars correspond to the outbound lines.

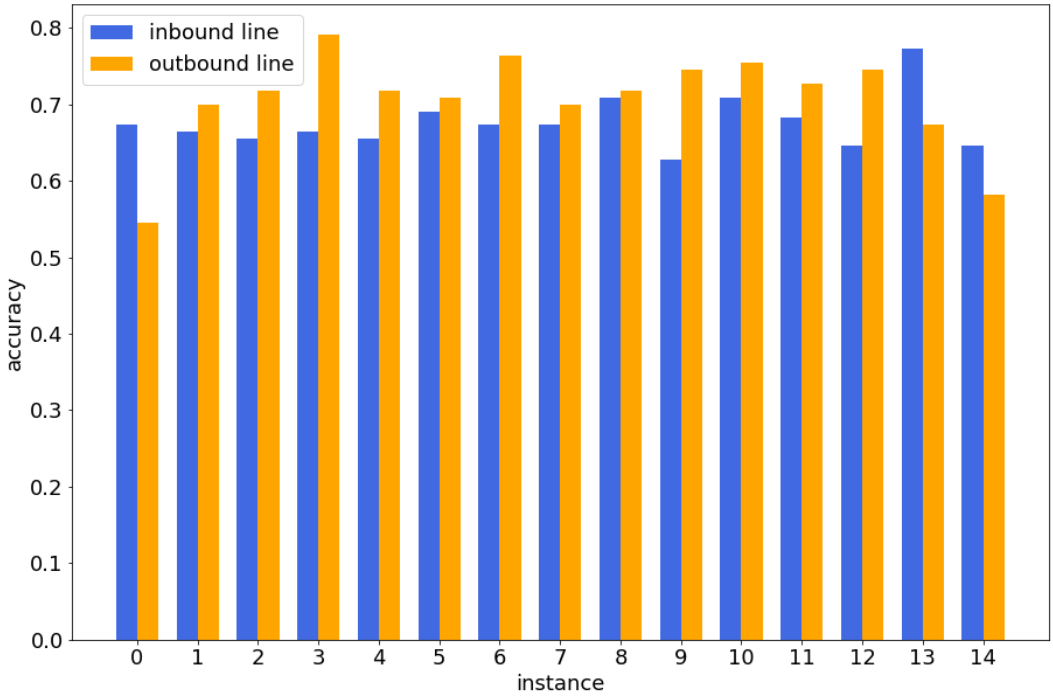


Figure 6.7: Prediction accuracy per instance for realistic BSP instances.

Results show an overall good prediction accuracy across all problem instances. For the inbound line classifier, the maximum prediction accuracy was 0.77, the minimum was 0.63, and the median was 0.67. The highest prediction accuracy for the outbound lines was 0.79, the minimum was 0.54, and the median was 0.72. These are promising results when considering the number of class labels (i.e., number of possible predictions) of each classifier: 42 for the inbound line predictor and 27 for the outbound line predictor. Thus, a random prediction over the possible values would render an average accuracy of 0.02 (inbound classifier) and 0.04 (outbound classifier).

6.3.4.3 Number of valid predictions

Figure 6.8 presents histograms of the number of valid predictions per line for each studied instance. Results show that most bus lines had a single valid prediction on all problem instances. In these cases, the predicted headways

were directly used for generating the candidate solutions. Some instances had lines with no valid predictions. No valid predictions could be made at most for two out of the 78 bus lines in each problem instance (instances #3, #6, and #9). In such cases, the headways assigned to that lines were randomly generated within the allowed ranges. Conversely, some lines have more than one valid prediction. The largest number of valid predictions for the same line in all problem instances was four (instances #2 and #4). In these cases, the headway for the line was randomly selected among the valid set of predictions when computing the candidate solutions for the problem.

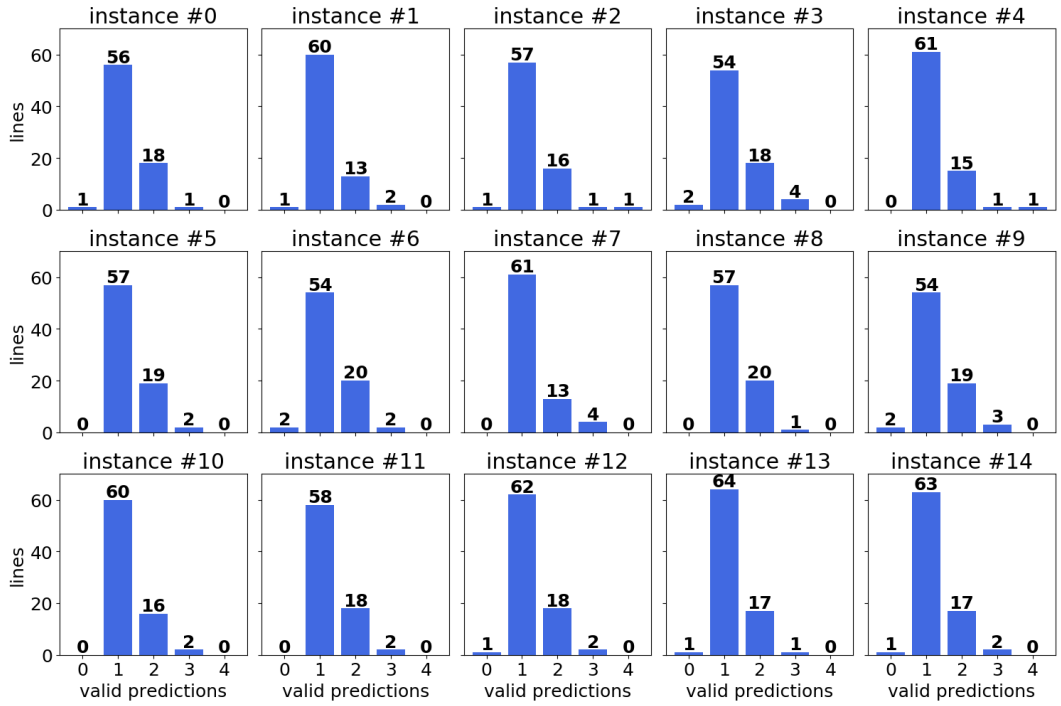


Figure 6.8: Number of valid predictions per line for realistic BSP instances.

6.3.4.4 Comparison to baseline solutions

VS was compared against the baseline solutions described in Section 6.3.2. Table 6.3 outlines the best results found by each algorithm in all 30 independent executions of each instance, except for h^i and real, which are deterministic. Results correspond to the score values computed using the procedure described in Algorithm 4. Results for three different configurations of VS are displayed: VS (0), corresponding to VS when using no improvement operator (i.e., only the prediction phase of VS); VS (1000), corresponding to VS with a 1000-step

LS improvement operator; and VS (5000), corresponding to VS with a 5000-step LS improvement operator. Score results are normalized using the score computed by the EA as reference to ease the comparison. The best result achieved on each problem instance is highlighted in gray. Raw results without normalization can be found in Table A.3 in Appendix A. Figure 6.9 shows boxplots of the score distribution over the 30 independent executions of each instance. For the sake of clarity, only the results computed by the different configurations of VS and the reference EA are displayed. Boxplots considering all the baseline solutions are reported in Figure A.11 in Appendix A.

Table 6.3: Results comparison of VS with baseline solutions for realistic BSP instances.

instance	*real*	h^i	*random*	*LS (1000)*	*EA*	*VS (0)*	*VS (1000)*	*VS (5000)*
0	0.9282	0.9757	0.9168	0.9939	1.0	0.9952	0.9987	0.9989
1	0.8816	0.9689	0.8835	0.9924	1.0	0.9964	0.9999	1.0002
2	0.9290	0.9765	0.9093	0.9927	1.0	0.9965	0.9990	0.9994
3	0.9161	0.9769	0.9090	0.9904	1.0	0.9877	0.9985	0.9994
4	0.9279	0.9749	0.9059	0.9950	1.0	0.9984	1.0003	1.0005
5	0.9170	0.9740	0.9182	0.9923	1.0	0.9984	1.0002	1.0008
6	0.8861	0.9695	0.8639	0.9896	1.0	0.9895	0.9993	1.0009
7	0.8973	0.9670	0.8787	0.9955	1.0	0.9958	0.9996	1.0004
8	0.8817	0.9647	0.8842	0.9907	1.0	0.9951	1.0014	1.0017
9	0.9165	0.9773	0.9027	0.9924	1.0	0.9907	0.9988	0.9996
10	0.9280	0.9750	0.9127	0.9955	1.0	0.9985	1.0004	1.0008
11	0.9170	0.9740	0.9142	0.9931	1.0	0.9976	0.9998	1.0003
12	0.9239	0.9747	0.9268	0.9906	1.0	0.9912	0.9999	1.0006
13	0.8883	0.9683	0.8707	0.9913	1.0	0.9933	1.0003	1.0010
14	0.9238	0.9746	0.9074	0.9921	1.0	0.9954	0.9996	1.0002

VS was able to compute accurate results even when considering only the prediction phase. VS (0), i.e., only the prediction phase of VS without any improvement operator, outperformed the real solution, i.e., the configuration of headways according to the public transportation timetable, by up to 11.5%. Additionally, VS (0) outperformed the random solution by up to 12.6% and the minimum headway solution by up to 3.0%. In the best case, the solution computed using only the prediction phase of VS was 99.8% as good as the solution computed by the reference EA. In median, the prediction phase computed solutions that were 99.5% as good as the reference solution and, in the worst case, the computed solution was 98.7% as good as the EA solution. These are highly competitive results, that demonstrate the capability of the prediction phase of VS to compute accurate solutions for the studied instances.

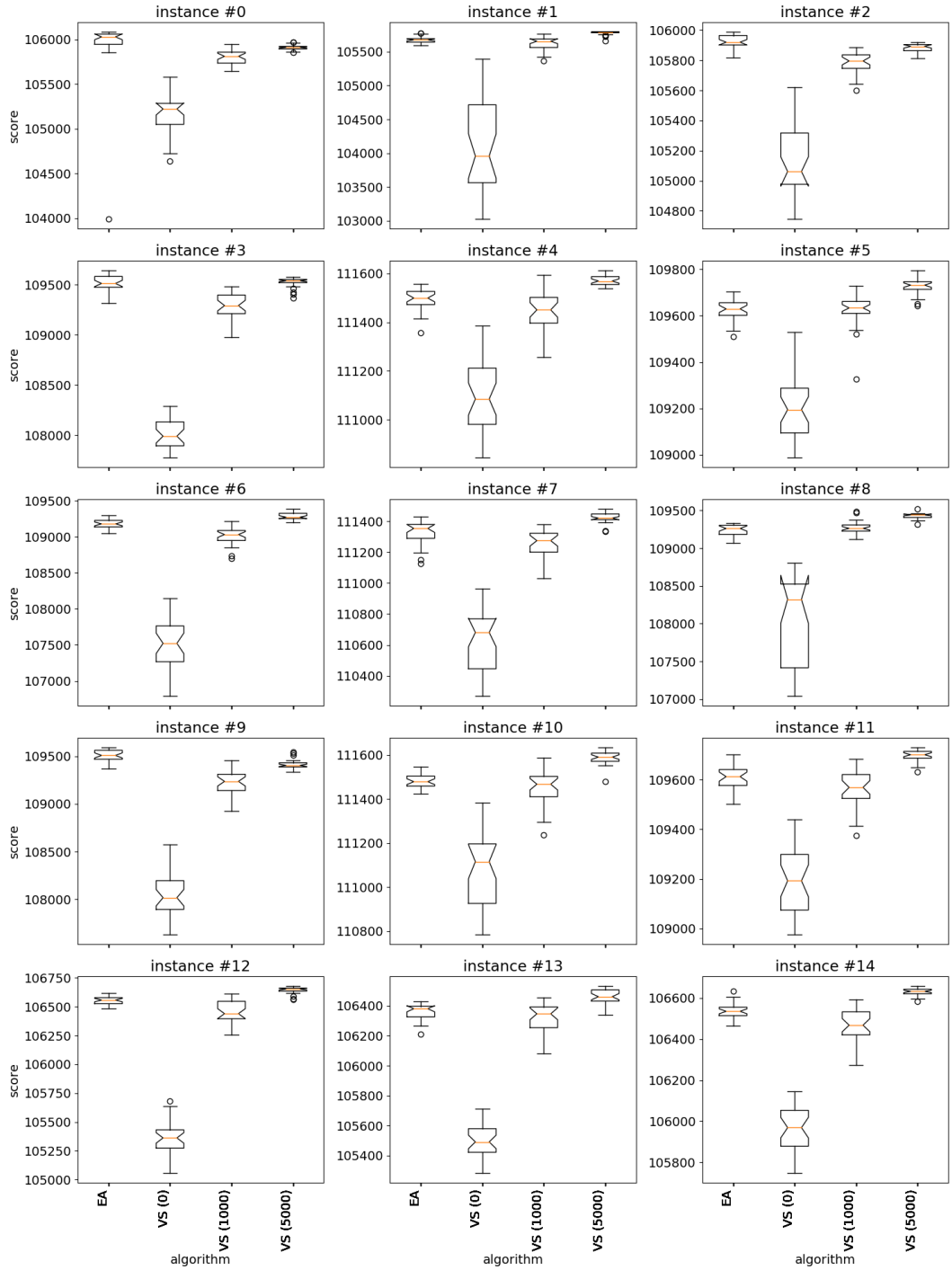


Figure 6.9: Results comparison of VS vs. EA on realistic BSP instances.

Including the LS improvement operator of VS allowed further refining the computed solutions. In median, adding a 1000-step LS allowed reaching solutions 0.37% closer to the reference computed by the EA compared to only using the prediction phase. When performing a LS of 5000 steps, the improvement increased up to 1.56% in the best case. With this configuration, VS outperformed the reference EA in eleven out of fifteen problem instances. Boxplots in Figure 6.9 show that the LS operator was also useful to reduce the variance of the computed results.

The contribution of each phase of VS is demonstrated when comparing the results of VS and the randomly generated solution with and without the LS operator. VS (0) outperformed the random solution by 8.80% in median. Similarly, VS (1000) outperformed LS (1000) on all studied problem instances. These results show that both phases of VS contributed to the overall result.

VS was trained with instances with up to 70 synchronization nodes and evaluated on instances with up to 110 nodes. The experimental results show that VS was able to effectively scale in the problem dimension, solving instances larger than those seen in the training phase.

6.4 Conclusions

This chapter presented the application of VS to solve the BSP, a complex, real-world, combinatorial optimization problem.

A new problem formulation was defined, extending previous models in the literature by considering transfers at any pair of bus stops in the public transportation system and accounting for the walking time between bus stops. The problem was solved with VS using RF as machine learning classifiers. VS was trained using an EA as a reference and several feature configurations were compared to improve the prediction accuracy of the proposed approach.

The experimental analysis was performed using two sets of problem instances: one comprised of 130 synthetic instances and the other of 45 realistic instances modeling the public transportation network in Montevideo, Uruguay. An EA was used as a reference algorithm to train VS and to evaluate its efficacy along with other baseline solutions. VS was able to compute accurate solutions in both sets of problem instances. In the synthetic dataset, VS computed solutions within 1.2% of the reference EA in median when only considering the prediction phase and within 0.2% in median when including a 5000-step LS im-

provement operator. In the realistic instances from Montevideo, VS computed solutions 99.5% as good as the reference EA in median when considering only the prediction phase and outperformed the EA in eleven out of fifteen problem instances when adding a 5000-step LS improvement operator.

The BSP allowed studying the applicability of the VS paradigm to a real-world optimization problem and evaluating its effectiveness with respect to baseline solutions. A more complex problem decomposition than those applied for the NRP and HCSP was needed to solve the BSP. The applied problem decomposition involved training two separate machine learning classifiers. Additionally, the implementation was done using RF—in contrast with the two previous applications of VS which used SVMs—showing the versatility and adaptability of VS. The experimental evaluation was performed over larger instances (in terms of the number of bus lines and synchronization nodes) than those considered in the training phase. The experimental results highlight the scalability properties of VS in terms of the problem dimension, which were also noted in the other problems addressed in this thesis.

Chapter 7

Conclusions and Future Work

This closing chapter outlines the main findings and conclusions from the research reported in this thesis, along with the main lines of future work.

7.1 Conclusions

This thesis explored VS, a paradigm inspired by the Savant Syndrome that combines machine learning and parallel computing to solve complex optimization problems. Firstly, a thorough review of the related works in the literature was performed, focusing on the automatic generation of parallel programs and on the application of machine learning to optimization. Then, implementations of VS were developed and evaluated for solving three optimization problems: i) the NRP, a well-known problem from software engineering that was modeled as a 0/1-KP; ii) the HCSP, a classic task scheduling problem relevant in modern computing infrastructures; and iii) the BSP, an optimization problem related to public transportation networks.

The VS implementation for the NRP used the Nemhauser-Ullmann algorithm as a reference, which computes exact solutions for the problem. A thorough analysis was done considering different feature configurations, training set sizes, and parameters for the SVMs used as classifiers. Five different improvement operators were evaluated over a large dataset of instances with varying size and difficulty. When considering only the prediction phase, VS reached median prediction accuracies over 90% when grouping instances by their size and larger than 80% when grouping instances by their difficulty. Out of the five improvement operators that were evaluated, the simplest one—

a greedy strategy—achieved the best results. With this improvement operator, VS computed solutions within 1% from the optima for all studied instances.

The first parallel implementation of VS was evaluated when solving the HCSP. In this case, VS used MinMin as a reference, which is a well-known greedy heuristic for the problem. Experimental results showed that VS was able to outperform MinMin in most of the 180 problem instances studied, with improvements of up to 15%. The performance of VS was evaluated on four different computing platforms. Results showed that, thanks to its massively-parallel design, VS was able to take advantage of available computing resources to find accurate solutions for the HCSP in both shared- and distributed-memory architectures. Finally, experiments on very large problem instances were performed, which showed the excellent scalability properties of VS when solving instances up to 128 times larger than those seen during training.

Lastly, VS was applied to the BSP, a complex combinatorial optimization problem arising in public transportation. In this case, VS used RF trained with the results from an EA. Experimental evaluation was performed over a set of 130 synthetic instances and a set of 45 realistic instances from the public transportation network in Montevideo, Uruguay. VS computed accurate solutions in both datasets, achieving results within 0.2% of the reference EA in median on the synthetic instances and outperforming the EA in eleven out of fifteen realistic instances.

Some general remarks can be made regarding VS, which were observed in the different applications studied.

- Due to its flexible design, VS can use different machine learning algorithms for the training and prediction phases. In the studied problems two different classifiers were used: SVMs for the NRP and HCSP; and RFs for the BSP. These examples showed the versatility and adaptability of VS in terms of the learning strategy.
- Similarly, the design of VS is flexible in terms of the algorithm(s) used as a reference. On the studied problems, both exact and approximate algorithms were used as a reference. For the NRP, the Nemhauser-Ullmann algorithm was used, which computes exact solutions. In turn, the VS implementation for the HCSP used MinMin as a reference, an approximate greedy heuristic. Finally, when solving the BSP, VS used

an EA that computed approximate solutions for the problem. The use of different algorithms as a reference showed the flexibility of VS, which can take advantage of any available solver for the problem at hand.

The scalability of VS was evaluated both in terms of the problem dimension and in the use of computational resources.

- Regarding the problem dimension, VS was evaluated over problem instances much larger than those seen during training. When solving the HCSP, VS was trained with instances up to 512×16 (tasks×machines) and evaluated on much larger instances in terms of tasks (up to 65536×16). For the synthetic BSP instances, VS was trained using instances generated over grids of up to 400×400 and evaluated on instances corresponding to grids of size 450×450. In the realistic BSP instances, VS was trained with instances with up to 70 synchronization nodes and evaluated on instances with up to 110 nodes. In all cases, VS was able to accurately scale in the problem dimension, computing high-quality solutions in instances much larger than those seen during training. This is a very interesting feature of the design of VS, since it allows solving problem instances that may not be tractable for the algorithm used as a reference.
- The scalability in the use of computational resources was evaluated on the HCSP. For this problem, four different computing platforms were considered, including shared- and distributed-memory architectures. Results showed that the massively-parallel design of VS allows efficiently using available computing resources. When increasing the number of computing units, VS was able to significantly reduce the times of the prediction phase. Moreover, the overall computation time did not increase despite the computational demand of VS increases with the number of available resources.

Summarizing, the main contributions of the research reported in this thesis are:

- A review of the related works on the automatic generation of parallel programs and the synergy between machine learning and optimization.
- The definition of the workflow of VS and its implementation details.

- The implementation and experimental evaluation of VS when solving the NRP, modeled as a 0/1-KP.
- The application of VS to solve the HCSP and its evaluation over four different parallel computing platforms.
- The implementation of VS for solving the BSP and its evaluation over synthetic and realistic instances.

The work reported in this thesis resulted in several publications including three journal articles (Massobrio et al., 2019; Massobrio and Nesmachnow, 2020; de la Torre et al., 2020) and six conference articles (Massobrio et al., 2016, 2018a,b,c; Nesmachnow et al., 2020; Massobrio et al., 2020). Additionally, two more articles were submitted and are currently under consideration for possible publication in the Applied Soft Computing journal.

7.2 Future Work

As noted in the literature review, the integration of machine learning into optimization algorithms is still at an early stage of development (Bengio et al., 2021). The work presented in this thesis was intended to be a step forward towards bringing closer the machine learning and optimization research fields, but many lines of work remain to be addressed.

Regarding the training and prediction phase of VS, other machine learning algorithms need to be considered. Despite two different classifiers were used in this thesis (SVMs and RFs), more algorithms should be integrated into VS. One promising line of work is to incorporate ensemble learning to VS. In this approach, different machine learning classifiers could be trained to form an ensemble. These classifiers could even be trained using different optimization algorithms as a reference or over different sets of solved instances. The final prediction can be made using a voting mechanism that considers the predictions of all the classifiers in the ensemble. An ensemble comprised of SVMs, RFs, and ANNs achieved promising preliminary results when solving the HCSP, but further experimental evaluation is needed.

Regarding the improvement phase of VS, other operators should be considered and evaluated. Throughout the studied problems many improvement operators were evaluated, including different greedy heuristics and LS algorithms. The design of VS allows including different improvement operators

seamlessly. The rationale in the implementations presented in this thesis was to use general optimization strategies during the improvement phase. However, tailored improvement operators that incorporate problem-specific techniques could be easily included in VS. This could be necessary for particularly complex problems, where general optimization strategies may not be able to provide quality solutions. Additionally, another interesting line of work would be to use the pool of candidate solutions generated in the prediction phase of VS to initialize a population-based metaheuristic, e.g., an EA.

Other optimization problems, potentially with harder constraints or dependencies between the problem variables, should also be considered. Throughout the process that led to this thesis, preliminary experiments were performed for other problems, including the Prize-collecting Steiner Tree Problem (Johnson et al., 2000) and the VRP (Ralphs et al., 2003). These problems could be further addressed, along with many other related optimization problems. In particular, problems including interactions or epistasis as hard constraints should be studied. In this regard, an interesting line of work related to the NRP, would be to consider other variants of the problem, which model dependencies among requirements. Several algorithms exist in the literature that deal with these dependencies by iteratively eliminating them and solving subproblems with no dependencies (Bagnall et al., 2001). Thus, an interesting line of work would be to incorporate VS into these iterative algorithms to solve NRP problem instances with dependencies.

Additionally, all the problems studied in this thesis correspond to combinatorial optimization problems (i.e., finding solutions within finite sets). Continuous optimization problems could also be addressed with VS. In this case, regression algorithms should be used instead of machine learning classifiers, since the prediction phase of VS should render real values for problem variables instead of classification labels.

Furthermore, addressing multiobjective optimization problems is an interesting line of future work. For this purpose, a domain decomposition approach could be implemented, using a linear combination of the objective functions and training a set of classifiers with different weights. In this way, each classifier would aim to predict a solution with a given trade-off between the problem objectives. In other words, each classifier will be focused on making predictions for a specific region of the Pareto front corresponding to the problem instance.

Finally, another line of future work corresponds to evaluating VS in other computational platforms. While many different platforms were considered within this thesis, the massively-parallel design of VS makes it suitable to many other architectures. For instance, VS could be evaluated in low-power clusters (e.g., comprised of Raspberry Pi or ARM processors) as well as in cloud infrastructures.

Bibliography

Aarts, E. and Lenstra, J. K. (2003). *Local search in combinatorial optimization*. Princeton University Press.

Araújo, A. A. and Paixão, M. (2014). Machine learning for user modeling in an interactive genetic algorithm for the next release problem. In *Search-Based Software Engineering*, pages 228–233, Cham. Springer International Publishing.

Araújo, A. A., Paixao, M., Yeltsin, I., Dantas, A., and Souza, J. (2016). An architecture based on interactive optimization and machine learning applied to the next release problem. *Automated Software Engineering*, 24(3):623–671.

Aurum, A. and Wohlin, C. (2005). *Engineering and Managing Software Requirements*. Springer-Verlag, Berlin, Heidelberg.

Ausiello, G., Crescenzi, P., Gambosi, G., Kann, V., Marchetti, A., and Protasi, M. (2012). *Complexity and approximation: Combinatorial optimization problems and their approximability properties*. Springer Science & Business Media.

Bagnall, A., Rayward, V., and Whittley, I. (2001). The next release problem. *Information and Software Technology*, 43(14):883–890.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations*.

Bello, I., Pham, H., Le, Q. V., Norouzi, M., and Bengio, S. (2017). Neural combinatorial optimization with reinforcement learning. In *5th International Conference on Learning Representations*.

Bengio, Y., Lodi, A., and Prouvost, A. (2021). Machine learning for combinatorial optimization: A methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421.

Bennett, K. P. and Parrado, E. (2006). The interplay of optimization and machine learning research. *Journal of Machine Learning Research*, 7(Jul):1265–1281.

Berthet, Q., Blondel, M., Teboul, O., Cuturi, M., Vert, J.-P., and Bach, F. (2020). Learning with differentiable perturbed optimizers. In *34th Conference on Neural Information Processing Systems*, pages 1–12.

Blum, C. and Roli, A. (2003). Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys*, 35(3):268–308.

Bottou, L., Curtis, F. E., and Nocedal, J. (2018). Optimization methods for large-scale machine learning. *Siam Review*, 60(2):223–311.

Braun, T., Siegel, H., Beck, N., Bölöni, L., Maheswaran, M., Reuther, A., Robertson, J., Theys, M., Yao, B., Hensgen, D., and Freund, R. (2001). A Comparison of Eleven Static Heuristics for Mapping a Class of Independent Tasks onto Heterogeneous Distributed Computing Systems. *Journal of Parallel and Distributed Computing*, 61(6):810–837.

Ceder, A., Golany, B., and Tal, O. (2001). Creating bus timetables with maximal synchronization. *Transportation Research Part A: Policy and Practice*, 35(10):913–928.

Ceder, A. and Tal, O. (1999). Timetable Synchronization for Buses. In *Lecture Notes in Economics and Mathematical Systems*, pages 245–258. Springer Berlin Heidelberg.

Ceder, A. and Wilson, N. (1986). Bus network design. *Transportation Research Part B: Methodological*, 20(4):331–344.

Chang, C.-C. and Lin, C.-J. (2011). LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27.

Cheang, S. M., Leung, K. S., and Lee, K. H. (2006). Genetic parallel programming: Design and implementation. *Evolutionary Computation*, 14(2):129–156.

Daduna, J. and Voß, S. (1995). Practical Experiences in Schedule Synchronization. In *Lecture Notes in Economics and Mathematical Systems*, volume 430, pages 39–55. Springer Berlin Heidelberg.

Dagum, L. and Menon, R. (1998). OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55.

Dantzig, G. B. (1957). Discrete Variable Extremum Problems. *Operations Research*, 5:266–277.

Darte, A., Robert, Y., and Vivien, F. (2012). *Scheduling and automatic parallelization*. Springer Science & Business Media.

de Castro, L. N. (2006). *Fundamentals of natural computing: basic concepts, algorithms, and applications*. CRC Press.

de la Torre, J. C., Massobrio, R., Ruiz, P., Nesmachnow, S., and Dorronsoro, B. (2020). Parallel virtual savant for the heterogeneous computing scheduling problem. *Journal of Computational Science*, 39:101048.

Dean, J. and Ghemawat, S. (2004). MapReduce: Simplied Data Processing on Large Clusters. In 6^{th} *Symposium on Operating System Design and Implementation*, pages 137–150.

Demirović, E., Stuckey, P. J., Bailey, J., Chan, J., Leckie, C., Ramamohanarao, K., and Guns, T. (2019). An investigation into prediction + optimisation for the knapsack problem. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 241–257, Cham. Springer International Publishing.

Dorronsoro, B., Nesmachnow, S., Taheri, J., Zomaya, A., Talbi, E.-G., and Bouvry, P. (2014). A hierarchical approach for energy-efficient scheduling of large workloads in multicore distributed systems. *Sustainable Computing: Informatics and Systems*, 4(4):252–261.

Dorronsoro, B. and Pinel, F. (2017). Combining Machine Learning and Genetic Algorithms to Solve the Independent Tasks Scheduling Problem. In *3rd IEEE International Conference on Cybernetics*, pages 1–8.

Duran, B. and Xhafa, F. (2006). The Effects of Two Replacement Strategies on a Genetic Algorithm for Scheduling Jobs on Computational Grids. In *ACM Symposium on Applied Computing*, pages 960–961.

Elmachtoub, A. N. and Grigas, P. (2017). Smart "predict, then optimize". *arXiv preprint arXiv:1710.08005*.

Fleurent, C., Lessard, R., and Séguin, L. (2004). Transit timetable synchronization: Evaluation and optimization. In *9th International Conference on Computer-aided Scheduling of Public Transport*.

Garey, M. R. and Johnson, D. S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA.

Golub, G. H. and Ortega, J. M. (2014). *Scientific computing: an introduction with parallel computing*. Elsevier.

Graham, S. L., Kessler, P. B., and Mckusick, M. K. (1982). Gprof: A Call Graph Execution Profiler. *SIGPLAN Notices*, 17(6):120–126.

Grava, S. (2002). *Urban Transportation Systems: Choices for Communities*. McGraw-Hill.

Grimes, D., Ifrim, G., O'Sullivan, B., and Simonis, H. (2014). Analyzing the impact of electricity price forecasting on energy cost-aware scheduling. *Sustainable Computing: Informatics and Systems*, 4(4):276–291.

Guyon, I., Gunn, S., Hur, A. B., and Dror, G. (2004). Result Analysis of the NIPS 2003 Feature Selection Challenge. In *17th International Conference on Neural Information Processing Systems*, pages 545–552.

Harman, M., Krinke, J., Medina-Bulo, I., Palomo, F., Ren, J., and Yoo, S. (2014). Exact scalable sensitivity analysis for the next release problem. *ACM Transactions on Software Engineering and Methodology*, 23(2):1–31.

Heaton, P. and Wallace, G. L. (2004). Annotation: The savant syndrome. *Journal of child psychology and psychiatry*, 45(5):899–911.

Hermelin, B. and O'Connor, N. (1990). Art and Accuracy: The Drawing Ability of Idiot-Savants. *Journal of Child Psychology and Psychiatry*, 31(2):217–228.

Hermelin, B., Pring, L., Buhler, M., Wolff, S., and Heaton, P. (1999). A Visually Impaired Savant Artist: Interacting Perceptual and Memory Representations. *Journal of Child Psychology and Psychiatry*, 40(7):1129–1139.

Hu, H., Zhang, X., Yan, X., Wang, L., and Xu, Y. (2017). Solving a New 3D Bin Packing Problem with Deep Reinforcement Learning Method. In *International Joint Conference on Artificial Intelligence-Workshop on AI application in E-commerce*, pages 1–7.

Hull, J. J. (1994). A database for handwritten text recognition research. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16(5):550–554.

Ibarra, O. and Kim, C. (1977). Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors. *Journal of the ACM*, 24(2):280–289.

Ibarra, O., López, F., and Rios, Y. (2016). Multiperiod Bus Timetabling. *Transportation Science*, 50(3):805–822.

Ibarra, O. and Rios, Y. (2012). Synchronization of bus timetabling. *Transportation Research Part B: Methodological*, 46(5):599–614.

Intel®Software (2012). Intel®Math Kernel Library Link Line Advisor. https://software.intel.com/en-us/articles/intel-mkl-link-line-advisor. Accessed: 2021-01-28.

International Transport Forum (2014). *Valuing Convenience in Public Transport*. OECD Publishing.

Irigoin, F., Jouvelot, P., and Triolet, R. (1991). Semantical Interprocedural Parallelization: An Overview of the PIPS Project. In 5^{th} *International Conference on Supercomputing*, pages 244–251.

Iturriaga, S., Nesmachnow, S., Luna, F., and Alba, E. (2014). A parallel local search in CPU/GPU for scheduling independent tasks on large heterogeneous computing systems. *The Journal of Supercomputing*, 71(2):648–672.

James, G., Witten, D., Hastie, T., and Tibshirani, R. (2014). *An Introduction to Statistical Learning: With Applications in R*. Springer Publishing Company.

Johnson, D. S., Minkoff, M., and Phillips, S. (2000). The Prize Collecting Steiner Tree Problem: Theory and Practice. In *11th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 760–769.

Jordan, M. I. and Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260.

Kellerer, H., Pferschy, U., and Pisinger, D. (2004). *Knapsack Problems*. Springer.

Khalil, E., Dai, H., Zhang, Y., Dilkina, B., and Song, L. (2017). Learning combinatorial optimization algorithms over graphs. In *Advances in neural information processing systems*, pages 6348–6358.

Khokhar, A., Prasanna, V., Shaaban, M., and Wang, C. (1993). Heterogeneous Computing: Challenges and Opportunities. *Computer*, 26(6).

Kogan, S., Levin, D., Routledge, B. R., Sagi, J. S., and Smith, N. A. (2009). Predicting risk from financial reports with regression. In *Human Language Technologies: Annual Conference of the North American Chapter of the Association for Computational Linguistics*, pages 272–280.

Kool, W., van Hoof, H., and Welling, M. (2019). Attention, learn to solve routing problems! In *7th International Conference on Learning Representations*.

Koza, J. R. (1992). *Genetic programming: on the programming of computers by means of natural selection*. MIT press.

Lichman, M. (2013). UCI Machine Learning Repository. http://archive.ics.uci.edu/ml. Accessed: 2021-01-28.

Luo, P., Lü, K., and Shi, Z. (2007). A revisit of fast greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 67(6):695–714.

Mandi, J., Stuckey, P. J., Guns, T., et al. (2020). Smart predict-and-optimize for hard combinatorial optimization problems. In *AAAI Conference on Artificial Intelligence*, volume 34, pages 1603–1610.

Martello, S. and Toth, P. (1990). *Knapsack Problems: Algorithms and Computer Implementations*. John Willey & Sons.

Massobrio, R. (2018). Urban mobility data analysis in Montevideo, Uruguay. M.Sc. thesis, Universidad de la República.

Massobrio, R., Dorronsoro, B., and Nesmachnow, S. (2018a). Virtual Savant for the Heterogeneous Computing Scheduling Problem. In *International Conference on High Performance Computing & Simulation*, pages 1–7.

Massobrio, R., Dorronsoro, B., and Nesmachnow, S. (2019). Virtual Savant for the Knapsack Problem: learning for automatic resource allocation. *Proceedings of the Institute for System Programming of the Russian Academy of Sciences*, 31(2):21–32.

Massobrio, R., Dorronsoro, B., Nesmachnow, S., and Palomo-Lozano, F. (2018b). Automatic program generation: Virtual Savant for the knapsack problem. In *International Workshop on Optimization and Learning: Challenges and Applications*, pages 1–2.

Massobrio, R., Dorronsoro, B., Palomo-Lozano, F., Nesmachnow, S., and Pinel, F. (2016). Generación automática de programas: Savant Virtual para el problema de la mochila. In *XI Congreso Español de Metaheurísticas, Algoritmos Evolutivos y Bioinspirados*, pages 1–10.

Massobrio, R. and Nesmachnow, S. (2020). Urban Mobility Data Analysis for Public Transportation Systems: A Case Study in Montevideo, Uruguay. *Applied Sciences*, 10(16):1–20.

Massobrio, R., Nesmachnow, S., and Dorronsoro, B. (2018c). Support Vector Machine Acceleration for Intel Xeon Phi Manycore Processors. In *High Performance Computing Latin America Conference*, pages 277–290.

Massobrio, R., Nesmachnow, S., and Dorronsoro, B. (2020). Virtual Savant: learning for optimization. In *34^{th} Conference on Neural Information Processing Systems. Learning Meets Combinatorial Algorithms workshop.*, pages 1–5.

Message Passing Interface Forum (2015). MPI: A Message-Passing Interface Standard. Version 3.1. Technical report, University of Tennessee. Accessed: 2021-01-28.

Midkiff, S. P. (2012). Automatic parallelization: An overview of fundamental compiler techniques. *Synthesis Lectures on Computer Architecture*, 7(1):1–169.

Mottron, L., Dawson, M., and Soulières, I. (2009). Enhanced perception in savant syndrome: patterns, structure and creativity. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 364(1522):1385–1391.

Mottron, L., Lemmens, K., Gagnon, L., and Seron, X. (2006). Non-Algorithmic Access to Calendar Information in a Calendar Calculator with Autism. *Journal of Autism and Developmental Disorders*, 36(2):239–247.

Nemhauser, G. L. and Ullmann, Z. (1969). Discrete Dynamic Programming and Capital Allocation. *Management Science*, 15(9):494–505.

Nemirovsky, D., Arkose, T., Markovic, N., Nemirovsky, M., Unsal, O., Cristal, A., and Valero, M. (2017). A Deep Learning Mapper (DLM) for Scheduling on Heterogeneous Systems. In *Latin American High Performance Computing Conference*, pages 3–20.

Nesmachnow, S. (2013). Parallel multiobjective evolutionary algorithms for batch scheduling in heterogeneous computing and grid systems. *Computational Optimization and Applications*, 55(2):515–544.

Nesmachnow, S., Baña, S., and Massobrio, R. (2017). A distributed platform for big data analysis in smart cities: combining Intelligent Transportation Systems and socioeconomic data for Montevideo, Uruguay. *EAI Endorsed Transactions on Smart Cities*, 2(5):1–18.

Nesmachnow, S., Cancela, H., and Alba, E. (2010). Heterogeneous computing scheduling with evolutionary algorithms. *Soft Computing*, 15(4):685–701.

Nesmachnow, S., Cancela, H., and Alba, E. (2012). A parallel micro evolutionary algorithm for heterogeneous computing and grid scheduling. *Applied Soft Computing*, 12(2):626–639.

Nesmachnow, S., Muraña, J., Goñi, G., Massobrio, R., and Tchernykh, A. (2020). Evolutionary Approach for Bus Synchronization. In *High Performance Computing Latin America Conference*, pages 320–336.

Norris, D. (1990). How to build a connectionist idiot (savant). *Cognition*, 35(3):277–291.

Nuseibeh, B. and Easterbrook, S. (2000). Requirements engineering. In *Conference on The Future of Software Engineering*. ACM Press.

Olariu, S. and Zomaya, A. Y. (2005). *Handbook of Bioinspired Algorithms and Applications*. CRC Press.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

Pinel, F. and Dorronsoro, B. (2014). Savant: Automatic generation of a parallel scheduling heuristic for map-reduce. *International Journal of Hybrid Intelligent Systems*, 11(4):287–302.

Pinel, F., Dorronsoro, B., and Bouvry, P. (2010). A new parallel asynchronous cellular genetic algorithm for scheduling in grids. In *International Symposium on Parallel & Distributed Processing*, pages 1–8.

Pinel, F., Dorronsoro, B., and Bouvry, P. (2018). The Virtual Savant: Automatic Generation of Parallel Solvers. *Information Sciences*, 432:411–430.

Pinel, F., Dorronsoro, B., Bouvry, P., and Khan, S. (2013). Savant: Automatic parallelization of a scheduling heuristic with machine learning. In *World Congress on Nature and Biologically Inspired Computing*, pages 52–57.

Pring, L. (2005). Savant talent. *Developmental medicine and child neurology*, 47(7):500–503.

Ralphs, T. K., Kopman, L., Pulleyblank, W. R., and Trotter, L. E. (2003). On the capacitated vehicle routing problem. *Mathematical programming*, 94(2-3):343–359.

Rimland, B. (1978). Savant capabilities of autistic children and their cognitive implications. pages 43–65. Brunner/Mazel.

Rimland, B. and Fein, D. (1988). Special talents of autistic savants. In *The exceptional brain: Neuropsychology of talent and special abilities*, pages 474–492. Guilford Press.

Russell, S. and Norvig, P. (2010). *Artificial intelligence: a modern approach*. Prentice Hall, 3 edition.

Ryan, C. and Ivan, L. (2000). Paragen – the first results. In *Genetic Programming*, pages 338–348. Springer Berlin Heidelberg.

Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., and Dill, D. L. (2019). Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*.

Snyder, A. (2009). Explaining and inducing savant skills: privileged access to lower level, less-processed information. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 364(1522):1399–1405.

Sodani, A., Gramunt, R., Corbal, J., Kim, H. S., Vinod, K., Chinthamani, S., Hutsell, S., Agarwal, R., and Liu, Y. C. (2016). Knights Landing: Second-Generation Intel Xeon Phi Product. *IEEE Micro*, 36(2):34–46.

Sra, S., Nowozin, S., and Wright, S. J. (2012). *Optimization for machine learning*. Mit Press.

Tournavitis, G., Wang, Z., Franke, B., and O'Boyle, M. F. (2009). Towards a holistic approach to auto-parallelization: Integrating profile-driven parallelism detection and machine-learning based mapping. In *30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 177–187.

Treffert, D. (2006). *Extraordinary People: Understanding Savant Syndrome*. iUniverse.

Treffert, D. (2009). The savant syndrome: an extraordinary condition. A synopsis: past, present, future. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 364(1522):1351–1357.

Treffert, D. A. (2013). Savant Syndrome: Realities, Myths and Misconceptions. *Journal of Autism and Developmental Disorders*, 44(3):564–571.

Veerapen, N., Ochoa, G., Harman, M., and Burke, E. K. (2015). An Integer Linear Programming approach to the single and bi-objective Next Release Problem. *Information and Software Technology*, 65:1–13.

Vinyals, O., Fortunato, M., and Jaitly, N. (2015). Pointer Networks. In *Advances in Neural Information Processing Systems 28*, pages 2692–2700.

Vlastelica, M., Paulus, A., Musil, V., Martius, G., and Rolínek, M. (2020). Differentiation of blackbox combinatorial solvers. In *International Conference on Learning Representations*.

Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., and Wang, Y. (2014). Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi™: How to Fully Exploit MIC Architectures*, pages 167–188. Springer International Publishing, Cham.

Wang, L., Siegel, H. J., Roychowdhury, V., and Maciejewski, A. A. (1997). Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach. *Journal of Parallel and Distributed Computing*, 47(1):8–22.

Wang, Y., Liu, H., Zheng, W., Xia, Y., Li, Y., Chen, P., Guo, K., and Xie, H. (2019). Multi-Objective Workflow Scheduling With Deep-Q-Network-Based Multi-Agent Reinforcement Learning. *IEEE Access*, 7:39974–39982.

Waschneck, B., Reichstaller, A., Belzner, L., Altenmüller, T., Bauernhansl, T., Knapp, A., and Kyek, A. (2018). Optimization of global production scheduling with deep reinforcement learning. *Procedia CIRP*, 72:1264–1269.

Weijters, A. J. M. M. (1995). The BP-SOM architecture and learning rule. *Neural Processing Letters*, 2(6):13–16.

Weijters, T., Van Den Bosch, A., and Postma, E. O. (2000). Learning statistically neutral tasks without expert guidance. In *Advances in Neural Information Processing Systems*, pages 73–79.

Wen, Y., Wang, Z., and Boyle, M. (2014). Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms. In *International Conference on High Performance Computing*.

Woeginger, G. J. (2003). *Exact Algorithms for NP-Hard Problems: A Survey*, pages 185–207. Springer Berlin Heidelberg.

Wu, T.-F., Lin, C.-J., and Weng, R. C. (2004). Probability estimates for multi-class classification by pairwise coupling. *Journal of Machine Learning Research*, 5(Aug):975–1005.

Xhafa, F. (2007). *A Hybrid Evolutionary Heuristic for Job Scheduling on Computational Grids*, pages 269–311. Springer Berlin Heidelberg.

Xhafa, F., Alba, E., Dorronsoro, B., and Duran, B. (2008). Efficient Batch Job Scheduling in Grids Using Cellular Memetic Algorithms. *Journal of Mathematical Modelling and Algorithms*, 7(2):217–236.

Xhafa, F., Carretero, J., Dorronsoro, B., and Alba, E. (2012). A tabu search algorithm for scheduling independent jobs in computational grids. *Computing and informatics*, 28(2):237–250.

Yamaguchi, M. (2009). Savant syndrome and prime numbers. *Polish Psychological Bulletin*, 40(2):69–73.

Zhao, G., Shen, Z., and Miao, C. (2009). ELM-based intelligent resource selection for grid scheduling. In *International Conference on Machine Learning and Applications*. IEEE.

APPENDICES

Appendix A

Supplementary results for the BSP

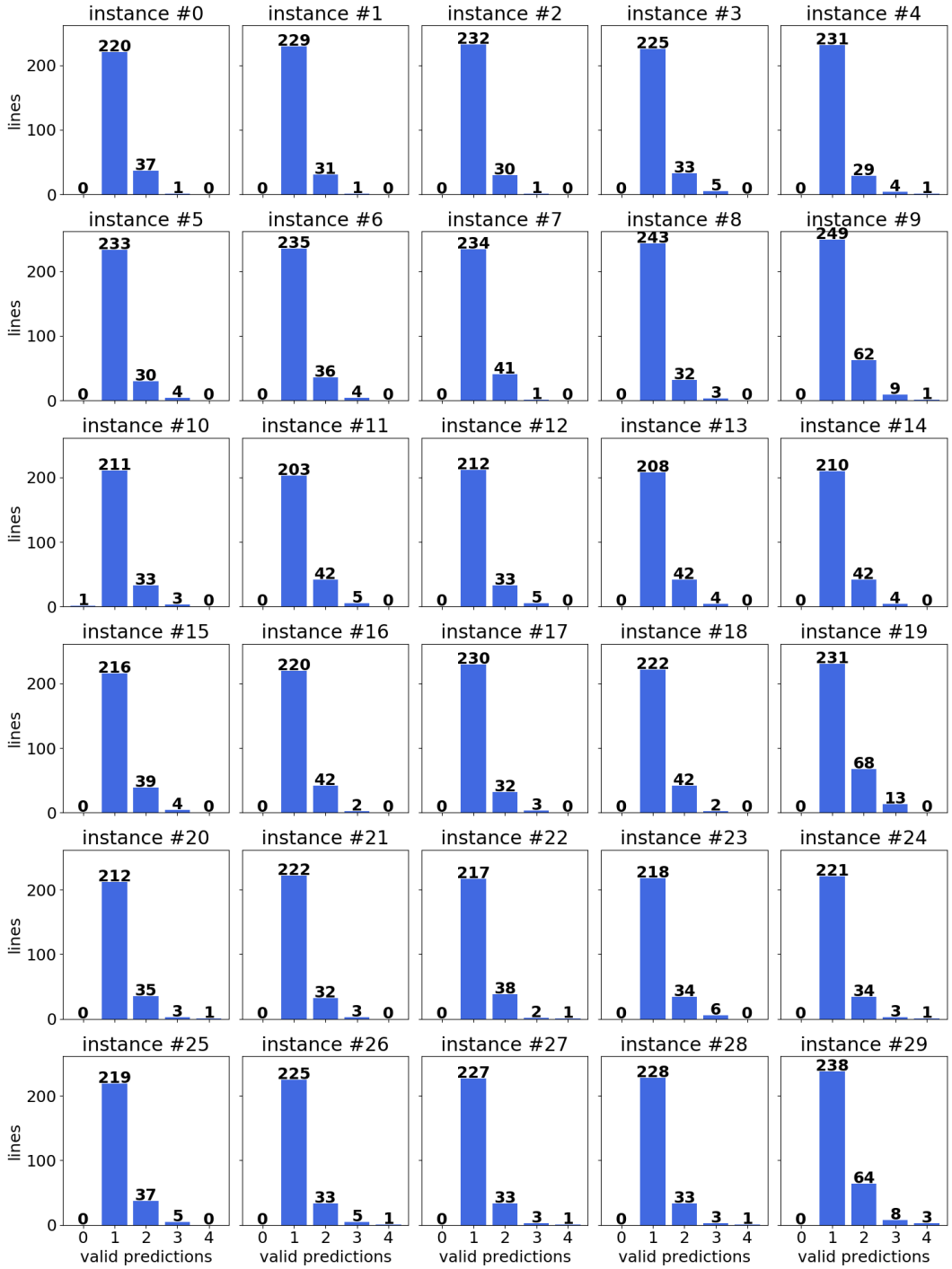


Figure A.1: Number of valid predictions per line for synthetic BSP instances 0-29.

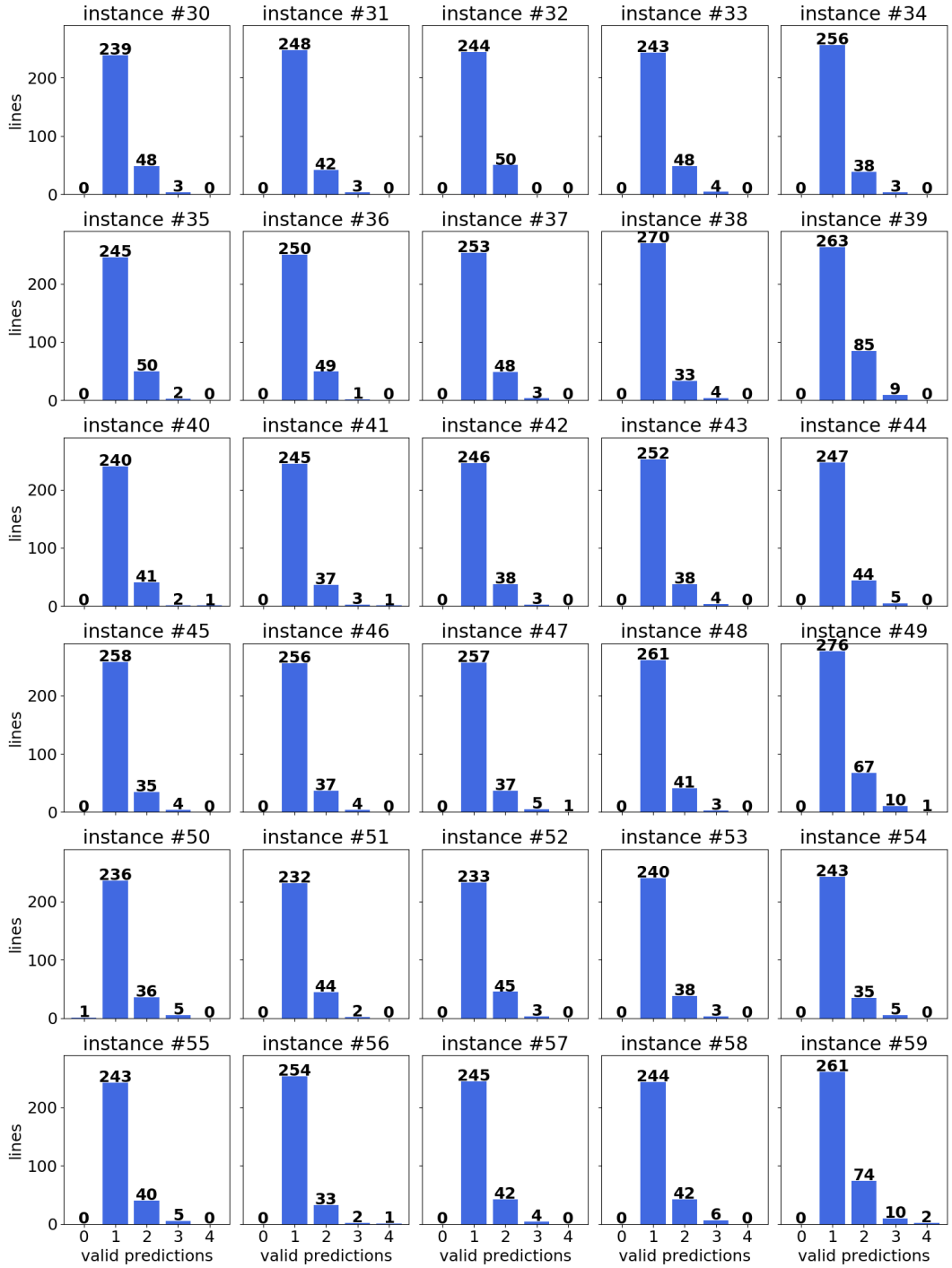


Figure A.2: Number of valid predictions per line for synthetic BSP instances 30-59.

Table A.1: Normalized results comparison of VS with baseline solutions for synthetic BSP instances.

instance	h^i	random	LS (1000)	EA	VS (0)	VS (1000)	VS (5000)
0	0.9622	0.7662	0.9266	1.0	0.9862	0.9919	0.9980
1	0.9645	0.7569	0.9297	1.0	0.9876	0.9926	0.9989
2	0.9637	0.7667	0.9258	1.0	0.9891	0.9930	0.9992
3	0.9636	0.7681	0.9264	1.0	0.9851	0.9914	0.9980
4	0.9603	0.7599	0.9324	1.0	0.9847	0.9910	0.9977
5	0.9621	0.7612	0.9225	1.0	0.9844	0.9897	0.9978
6	0.9602	0.7536	0.9195	1.0	0.9861	0.9924	0.9977
7	0.9608	0.7592	0.9193	1.0	0.9858	0.9920	0.9999
8	0.9583	0.7570	0.9231	1.0	0.9883	0.9932	0.9996
9	0.9652	0.7586	0.9153	1.0	0.9877	0.9925	0.9976
10	0.9693	0.7353	0.9254	1.0	0.9856	0.9917	0.9988
11	0.9681	0.7599	0.9263	1.0	0.9875	0.9926	0.9987
12	0.9722	0.7459	0.9227	1.0	0.9879	0.9929	0.9985
13	0.9652	0.7356	0.9172	1.0	0.9843	0.9898	0.9975
14	0.9703	0.7475	0.9262	1.0	0.9849	0.9914	0.9979
15	0.9701	0.7548	0.9274	1.0	0.9854	0.9925	0.9978
16	0.9707	0.7516	0.9142	1.0	0.9848	0.9893	0.9972
17	0.9702	0.7230	0.9053	1.0	0.9874	0.9934	0.9978
18	0.9704	0.7585	0.9242	1.0	0.9847	0.9908	0.9986
19	0.9732	0.7507	0.9055	1.0	0.9887	0.9920	0.9970
20	0.9703	0.7644	0.9273	1.0	0.9883	0.9936	0.9985
21	0.9689	0.7800	0.9396	1.0	0.9886	0.9922	0.9975
22	0.9685	0.7686	0.9261	1.0	0.9886	0.9931	0.9986
23	0.9668	0.7872	0.9281	1.0	0.9883	0.9930	0.9997
24	0.9667	0.7534	0.9170	1.0	0.9882	0.9928	0.9975
25	0.9695	0.7656	0.9309	1.0	0.9889	0.9929	0.9993
26	0.9659	0.7435	0.9142	1.0	0.9887	0.9936	0.9999
27	0.9677	0.7918	0.9324	1.0	0.9884	0.9930	0.9991
28	0.9672	0.7478	0.9213	1.0	0.9876	0.9924	0.9977
29	0.9701	0.7519	0.9045	1.0	0.9896	0.9926	0.9983
30	0.9648	0.7474	0.9089	1.0	0.9899	0.9933	0.9981
31	0.9683	0.7533	0.9139	1.0	0.9898	0.9938	0.9991
32	0.9684	0.7428	0.9104	1.0	0.9881	0.9927	0.9979
33	0.9642	0.7604	0.9064	1.0	0.9895	0.9928	0.9979
34	0.9680	0.7584	0.9058	1.0	0.9901	0.9931	0.9977
35	0.9687	0.7371	0.9063	1.0	0.9890	0.9936	0.9986
36	0.9667	0.7447	0.9095	1.0	0.9875	0.9915	0.9979
37	0.9630	0.7501	0.9060	1.0	0.9885	0.9927	0.9988
38	0.9688	0.7497	0.9077	1.0	0.9896	0.9926	0.9977
39	0.9695	0.7371	0.8955	1.0	0.9913	0.9941	0.9979
40	0.9731	0.7782	0.9153	1.0	0.9898	0.9937	0.9992
41	0.9729	0.7485	0.9172	1.0	0.9880	0.9927	0.9985
42	0.9713	0.7529	0.9152	1.0	0.9896	0.9931	0.9978
43	0.9720	0.7342	0.9123	1.0	0.9881	0.9916	0.9976
44	0.9731	0.7367	0.9132	1.0	0.9889	0.9927	0.9990
45	0.9726	0.7532	0.9144	1.0	0.9890	0.9932	0.9989
46	0.9708	0.7533	0.9066	1.0	0.9872	0.9915	0.9975
47	0.9733	0.7582	0.9070	1.0	0.9870	0.9912	0.9980
48	0.9723	0.7634	0.9130	1.0	0.9884	0.9930	0.9986
49	0.9753	0.7366	0.8904	1.0	0.9912	0.9937	0.9980
50	0.9692	0.7811	0.9206	1.0	0.9871	0.9924	0.9979
51	0.9707	0.7749	0.9244	1.0	0.9894	0.9918	0.9982
52	0.9683	0.7548	0.9192	1.0	0.9864	0.9925	0.9977
53	0.9704	0.7770	0.9282	1.0	0.9886	0.9937	0.9989
54	0.9674	0.7768	0.9246	1.0	0.9862	0.9916	0.9974
55	0.9652	0.7655	0.9145	1.0	0.9891	0.9927	0.9978
56	0.9738	0.7707	0.9182	1.0	0.9883	0.9923	0.9977
57	0.9705	0.7721	0.9285	1.0	0.9897	0.9939	0.9989
58	0.9661	0.7592	0.9243	1.0	0.9878	0.9923	0.9987
59	0.9722	0.7634	0.9008	1.0	0.9894	0.9935	0.9984

Table A.2: Results comparison of VS with baseline solutions for synthetic BSP instances.

instance	h^i	random	LS (1000)	EA	VS (0)	VS (1000)	VS (5000)
0	623085	496133	599991	647532	638609	642318	646228
1	591907	464547	570557	613722	606098	609166	613061
2	622024	494860	597517	645436	638427	640892	644934
3	614297	489664	590570	637515	628021	632045	636232
4	605774	479364	588121	630793	621120	625118	629362
5	602492	476689	577661	626200	616425	619769	624800
6	605176	475014	579584	630293	621538	625504	628834
7	595368	470404	569614	619640	610840	614659	619585
8	606072	478727	583815	632442	625067	628127	632211
9	1207527	949017	1145098	1251046	1235634	1241699	1248012
10	622940	472583	594736	642674	633390	637318	641902
11	640029	502400	612407	661115	652869	656208	660256
12	624506	479144	592690	642355	634573	637819	641384
13	598008	455747	568284	619590	609884	613281	618036
14	583853	449824	557317	601749	592685	596545	600512
15	610285	474840	583454	629107	619891	624370	627726
16	628341	486523	591811	647319	637468	640411	645508
17	613276	456995	572232	632111	624177	627969	630705
18	614587	480386	585317	633344	623636	627495	632435
19	1228219	947433	1142780	1262057	1247776	1252017	1258313
20	654587	515714	625599	674631	666738	670328	673640
21	668724	538354	648524	690202	682355	684810	688498
22	634243	503354	606486	654889	647450	650348	653949
23	625251	509084	600184	646714	639153	642207	646548
24	652202	508248	618652	674641	666684	669808	672949
25	643006	507778	617405	663240	655850	658555	662770
26	640473	492973	606208	663077	655609	658862	663036
27	639894	523538	616532	661233	653545	656636	660614
28	644504	498311	613875	666349	658087	661255	664840
29	1269704	984032	1183745	1308782	1295174	1299162	1306592
30	702182	543992	661515	727829	720454	722920	726466
31	717652	558338	677373	741154	733602	736535	740469
32	655071	502515	615850	676472	668407	671516	675065
33	709386	559450	666847	735717	727983	730407	734152
34	695859	545166	651140	718837	711688	713872	717207
35	704003	535672	658636	726763	718797	722102	725725
36	697515	537328	656205	721527	712519	715417	720029
37	664395	517523	625121	689950	682013	684888	689130
38	707718	547654	663067	730482	722854	725083	728830
39	1388780	1055887	1282788	1432528	1420003	1424038	1429525
40	676712	541117	636506	695387	688273	691004	694809
41	669879	515391	631546	688565	680284	683512	687541
42	665742	516084	627323	685429	678313	680710	683953
43	659170	497894	618718	678164	670097	672490	676547
44	688268	521079	645945	707312	699493	702175	706623
45	680641	527118	639925	699801	692124	695055	699047
46	674399	523294	629827	694692	685766	688811	692925
47	667319	519819	621899	685635	676731	679630	684252
48	662234	519912	621810	681090	673191	676332	680134
49	1346139	1016753	1229040	1380275	1368121	1371572	1377460
50	680834	548719	646702	702504	693458	697160	701009
51	701848	560242	668359	723000	715304	717063	721711
52	696982	543326	661643	719834	710048	714411	718214
53	659819	528350	631161	679954	672232	675675	679187
54	657969	528329	628838	680127	670740	674429	678378
55	685744	543899	649747	710498	702774	705335	708963
56	685289	542391	646144	703743	695509	698357	702143
57	673788	536062	644638	694286	687161	690061	693544
58	682718	536500	653170	706701	698055	701240	705764
59	1360419	1068345	1260600	1399378	1384531	1390261	1397183

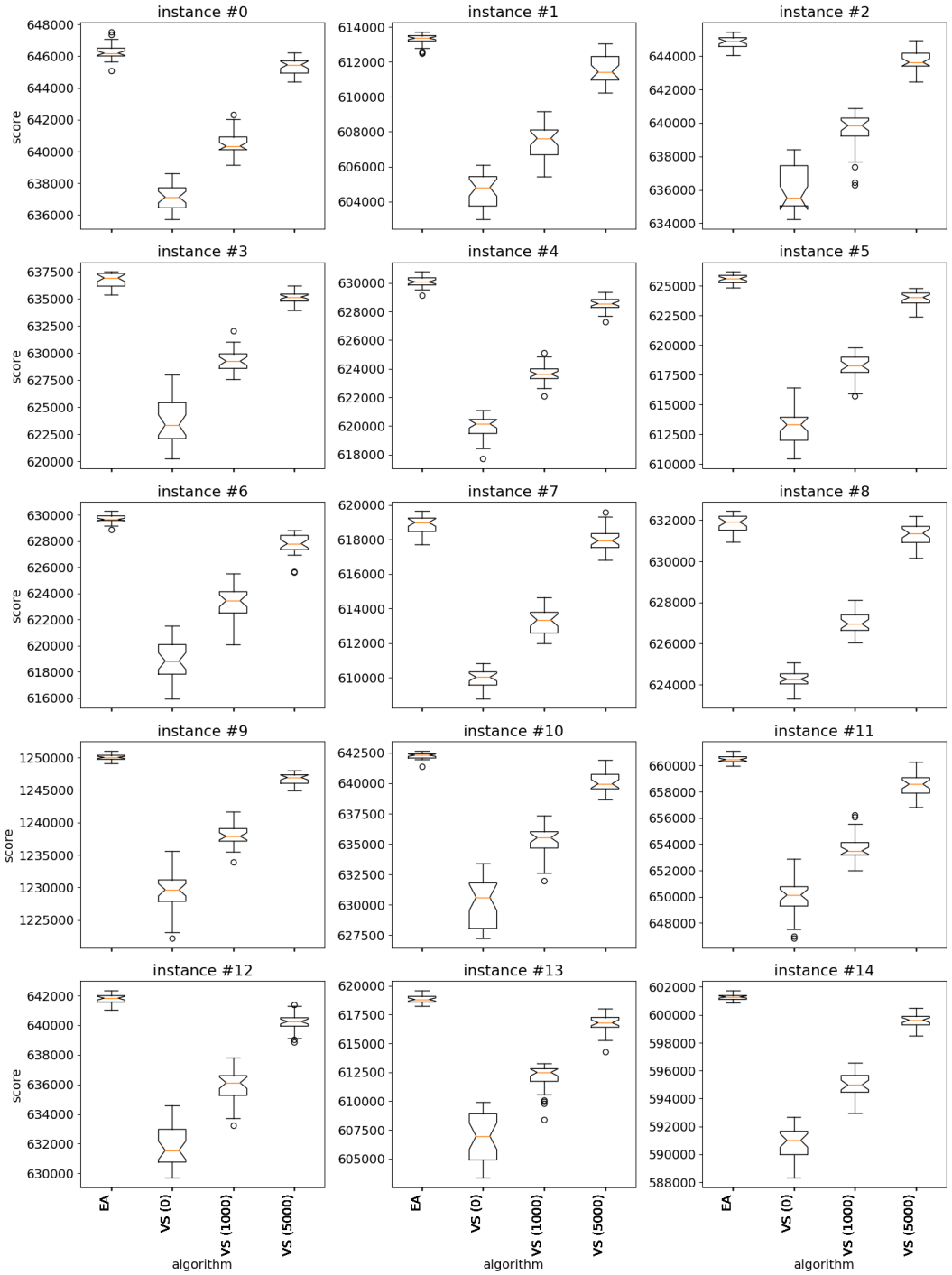


Figure A.3: Results comparison of VS vs. EA on synthetic BSP instances 0 to 14.

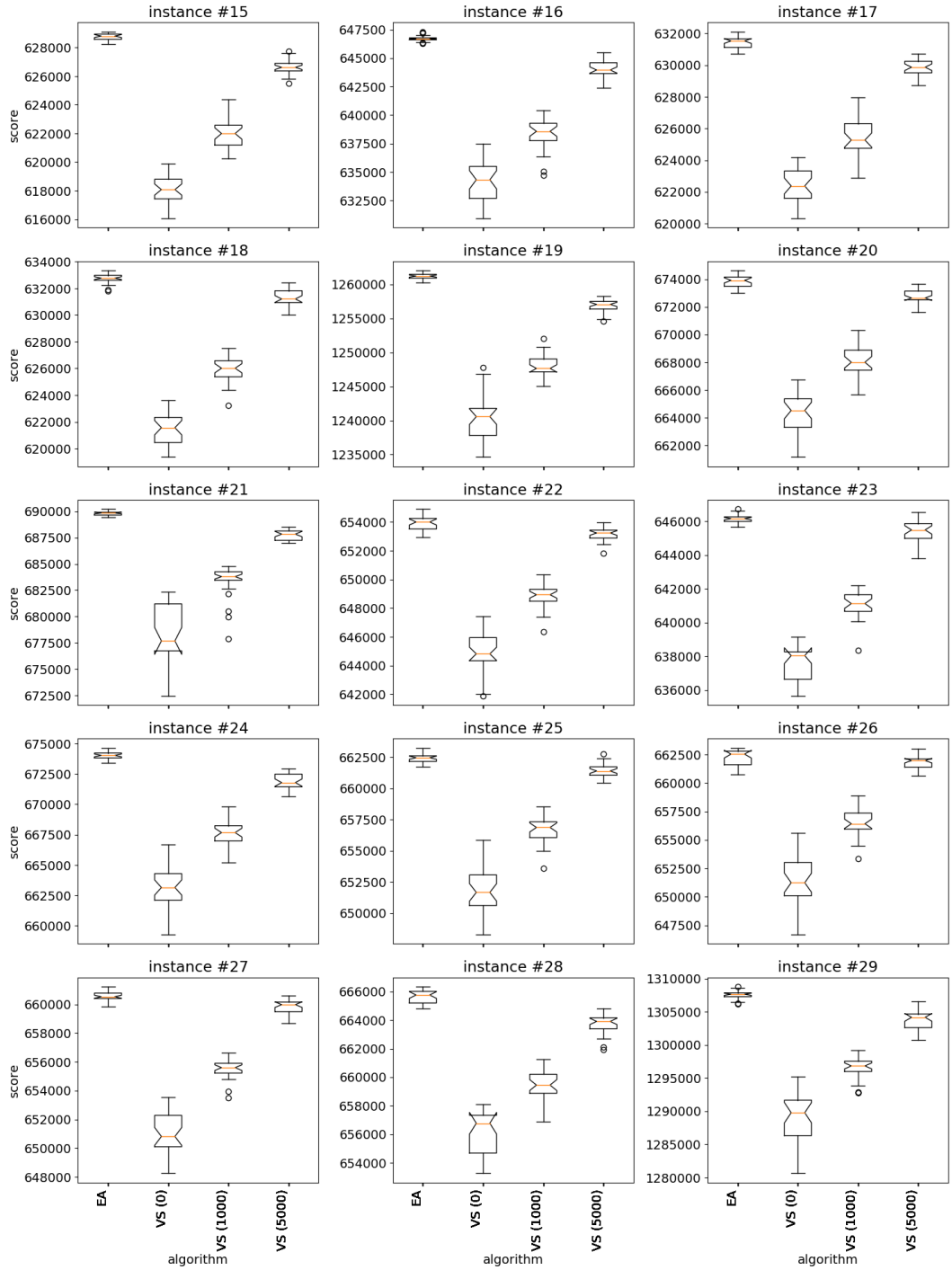


Figure A.4: Results comparison of VS vs. EA on synthetic BSP instances 15 to 29.

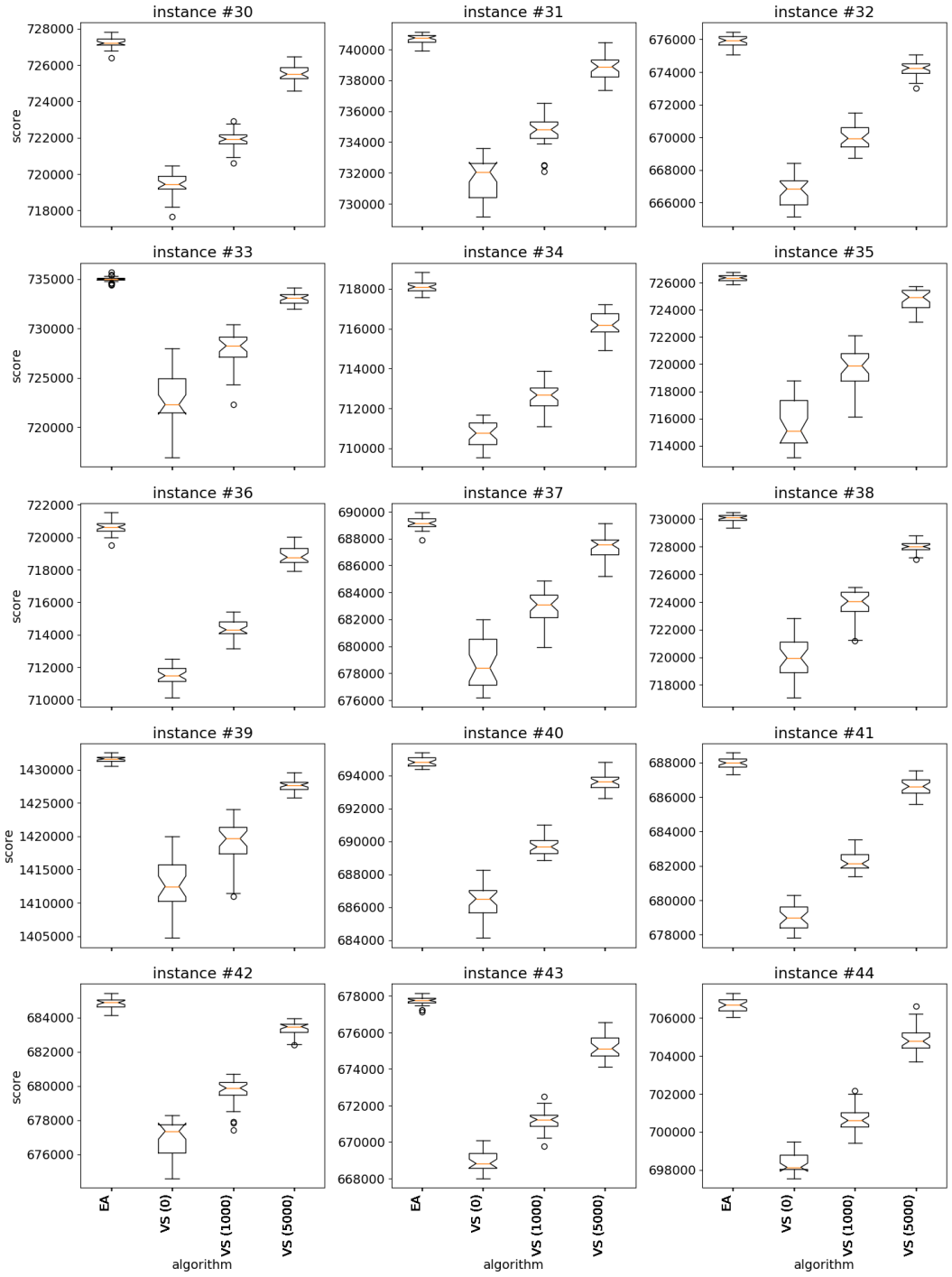


Figure A.5: Results comparison of VS vs. EA on synthetic BSP instances 30 to 44.

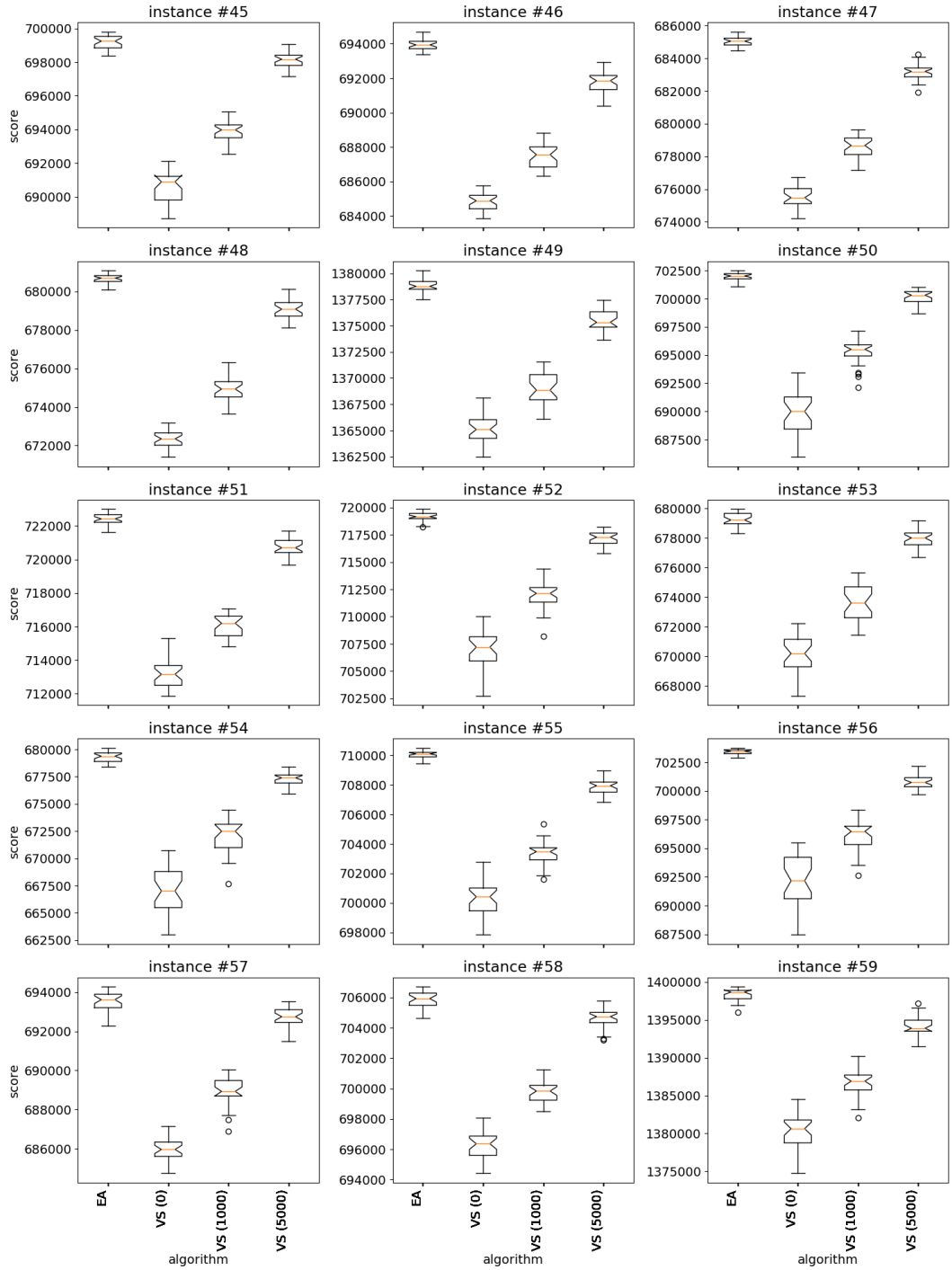


Figure A.6: Results comparison of VS vs. EA on synthetic BSP instances 45 to 59.

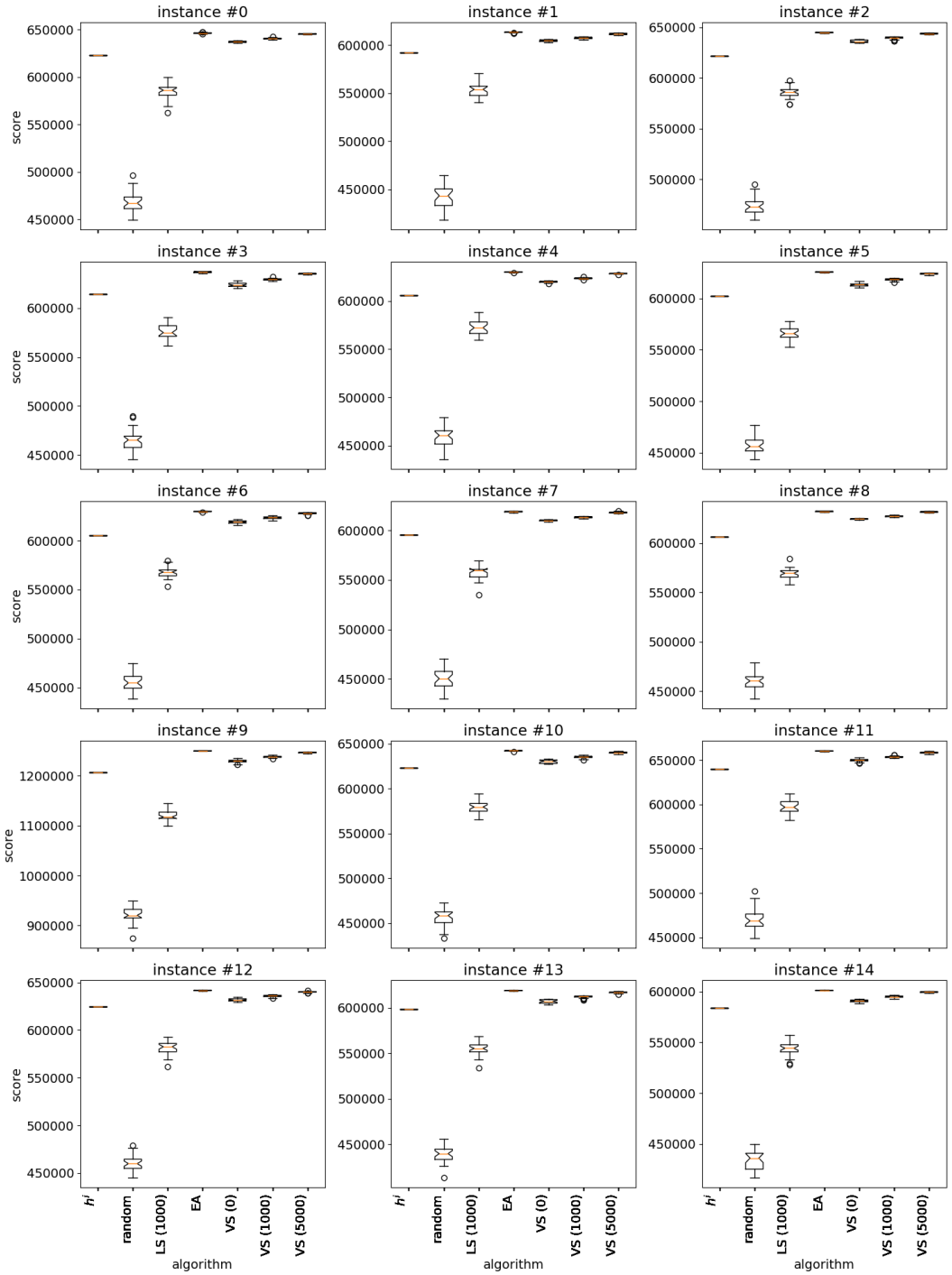


Figure A.7: Results comparison of VS with baseline solutions for synthetic BSP instances 0 to 14.

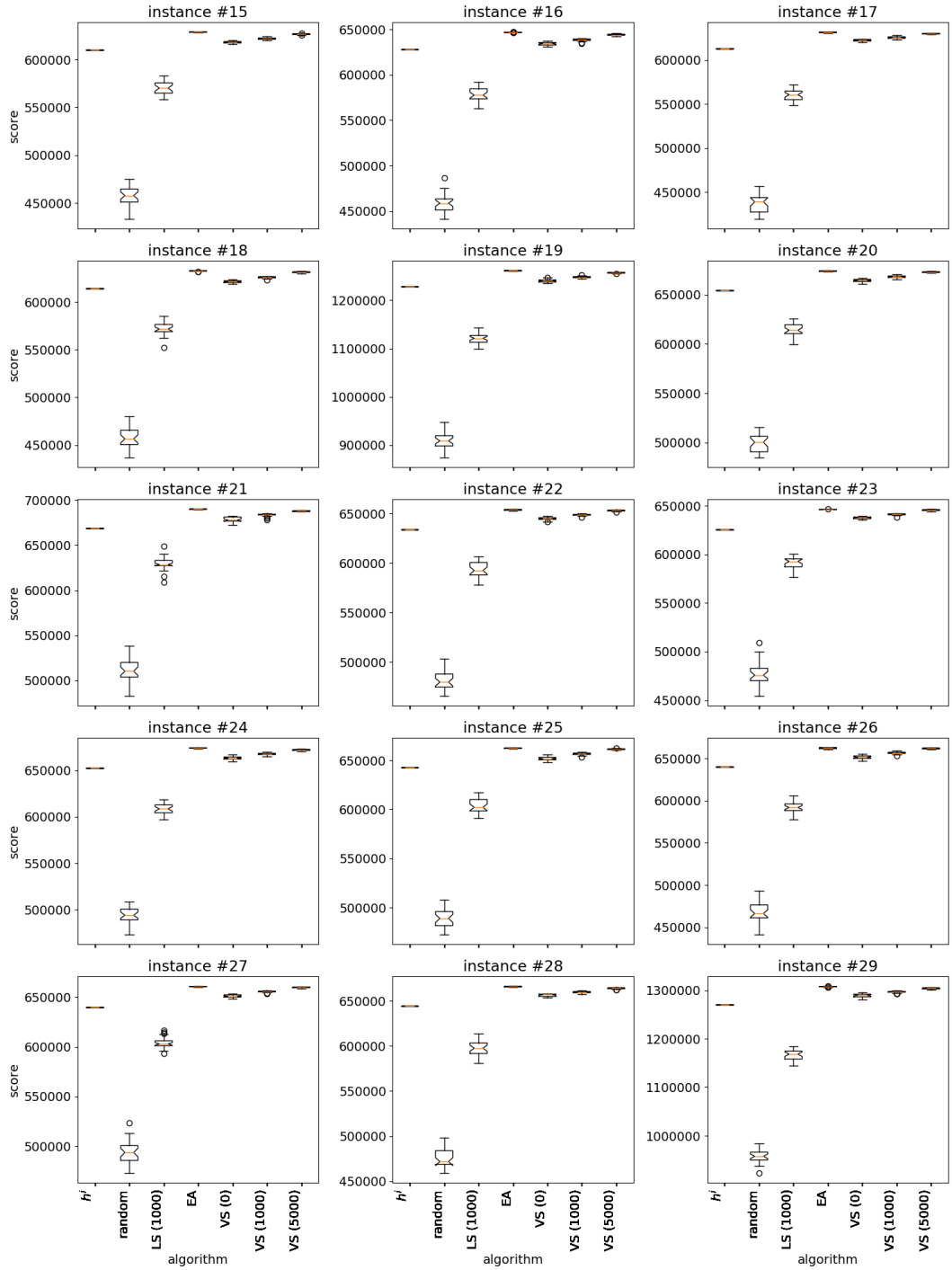


Figure A.8: Results comparison of VS with baseline solutions for synthetic BSP instances 15 to 29.

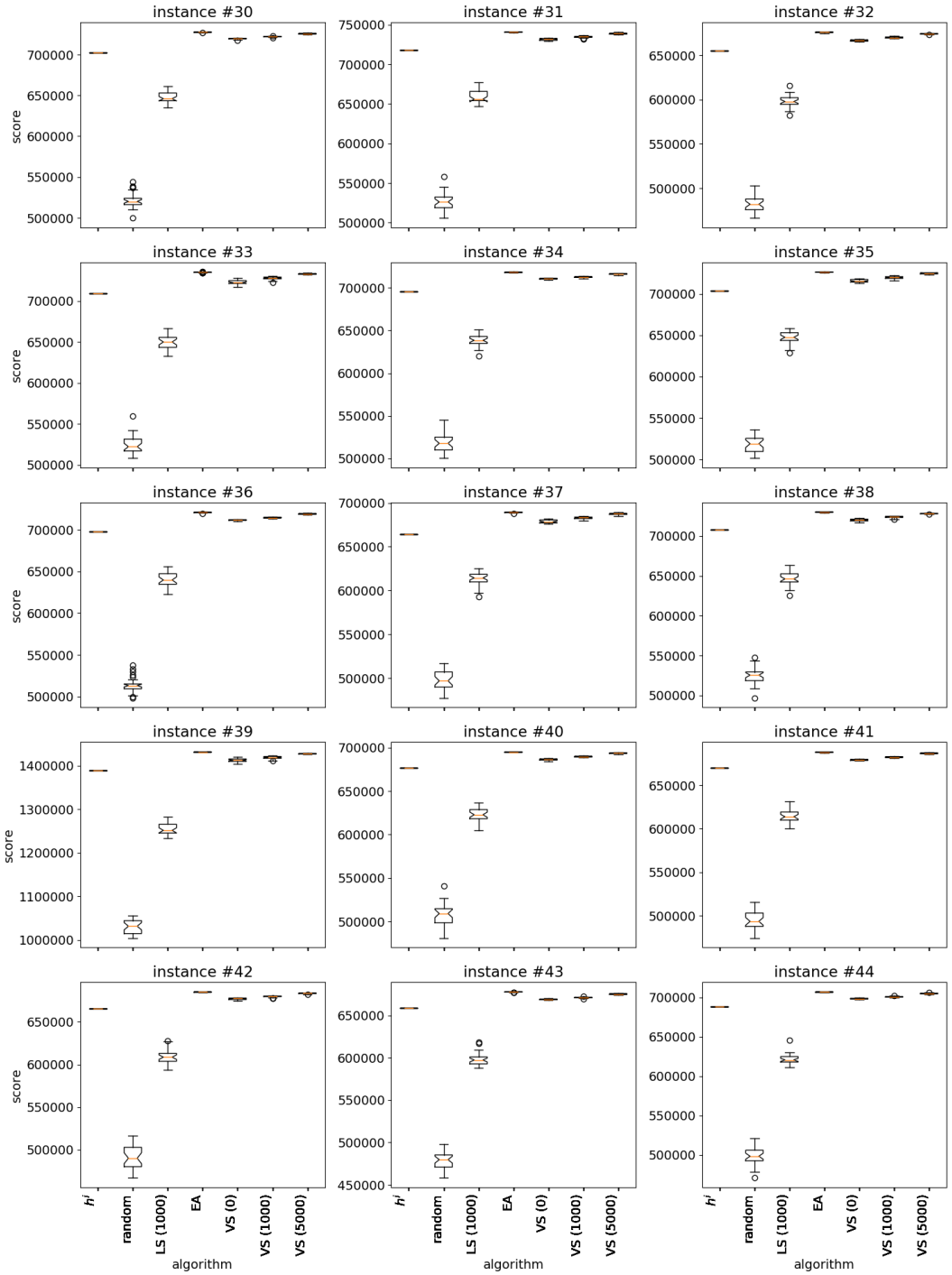


Figure A.9: Results comparison of VS with baseline solutions for synthetic BSP instances 30 to 44.

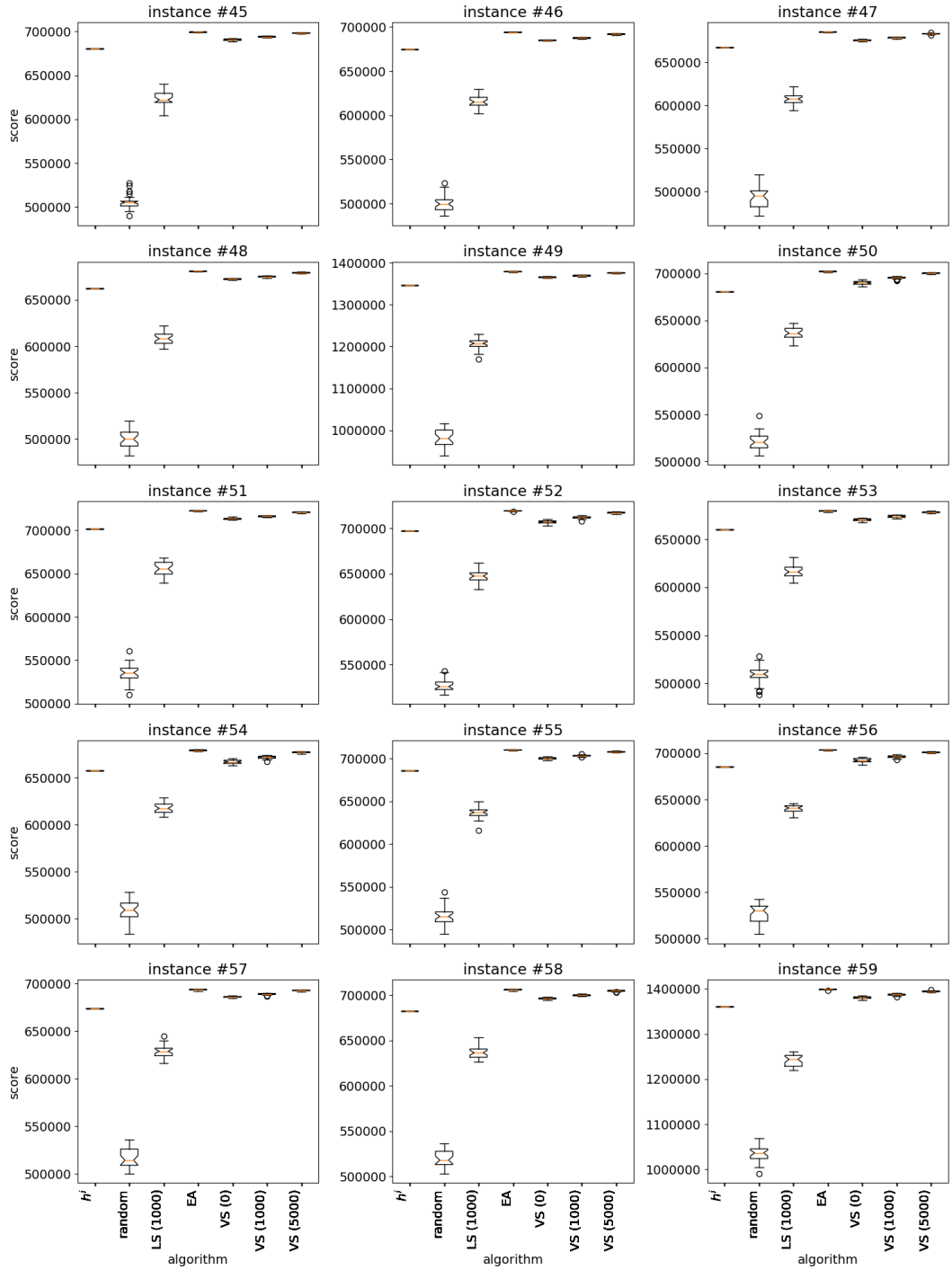


Figure A.10: Results comparison of VS with baseline solutions for synthetic BSP instances 45 to 59.

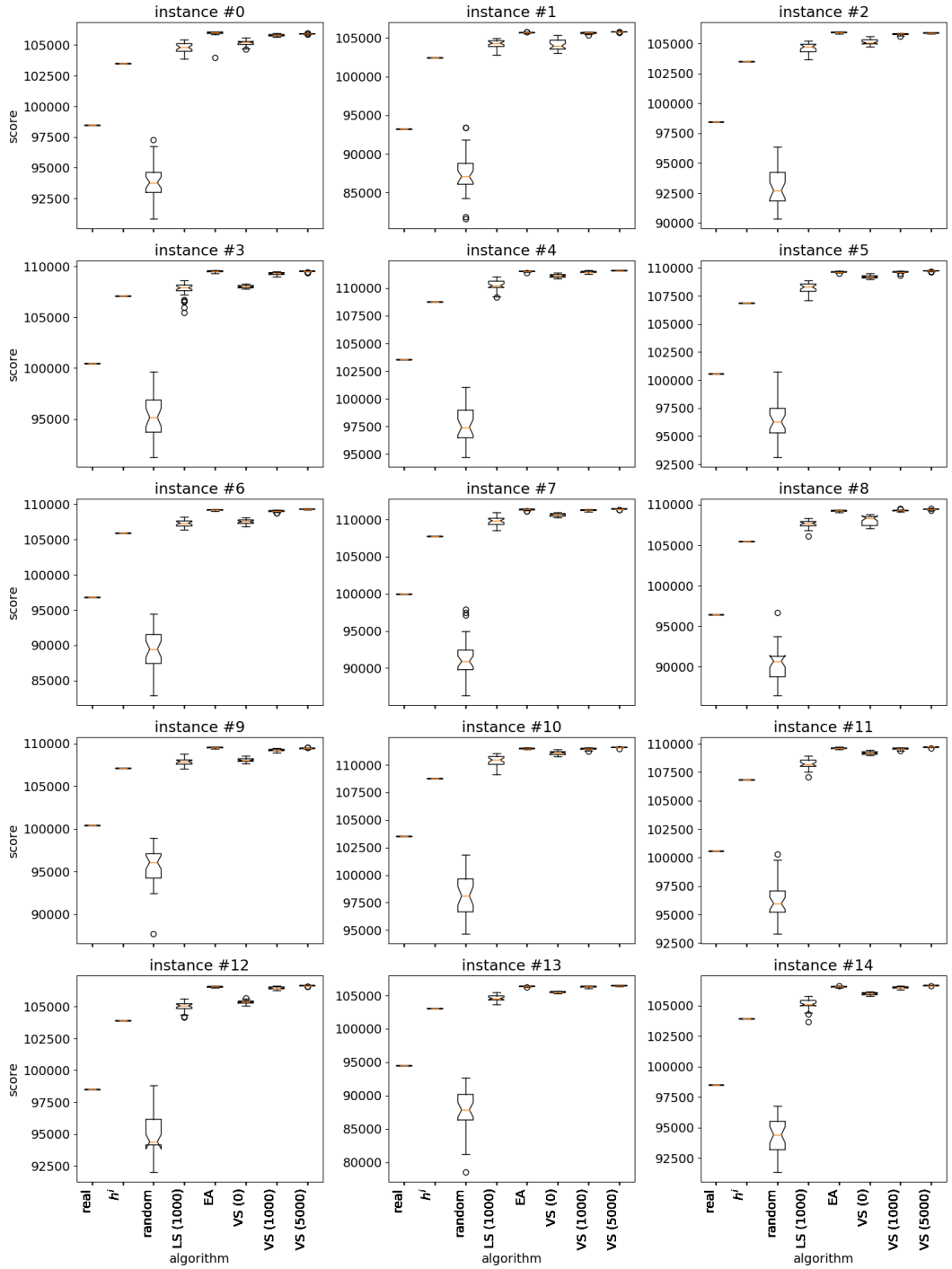


Figure A.11: Results comparison of VS with baseline solutions for realistic BSP instances.

Table A.3: Results comparison of VS with baseline solutions for realistic BSP instances.

instance	real	h^i	random	LS (1000)	EA	VS (0)	VS (1000)	VS (5000)
0	98467	103502	97260	105439	106085	105580	105949	105971
1	93248	102483	93451	104972	105772	105395	105764	105794
2	98467	103502	96381	105212	105989	105623	105885	105921
3	100442	107105	99662	108590	109642	108292	109483	109574
4	103520	108761	101057	110997	111558	111385	111593	111613
5	100601	106847	100726	108856	109703	109529	109729	109796
6	96848	105960	94423	108158	109293	108143	109220	109386
7	99988	107749	97917	110933	111429	110963	111379	111479
8	96403	105478	96672	108320	109332	108800	109481	109520
9	100442	107105	98928	108762	109592	108573	109459	109544
10	103520	108761	101814	111043	111547	111382	111587	111634
11	100601	106847	100288	108943	109703	109440	109684	109731
12	98507	103925	98816	105617	106620	105684	106612	106679
13	94543	103049	92664	105499	106427	105714	106455	106532
14	98507	103925	96759	105791	106633	106145	106592	106659